Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Kristian Zupan

Primerjava izvedb kriptografskih algoritmov na CPE in GPE

magistrsko delo

študijski program druge stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana, 2015

Izjava o avtorstvu magistrskega dela

Spodaj podpisani Kristian Zupan, z vpisno številko **63090064**, sem avtor magistrskega dela z naslovom:

Primerjava izvedb kriptografskih algoritmov na CPE in GPE

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela v zbirki "Dela FRI".

V Ljubljani, 7. junija 2015 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za vodenje, vzpodbujanje in pomoč pri izdelavi magistrske naloge. Zahvalil bi se tudi družini, Karmen ter vsem ostalim, ki so mi v tem času nudili podporo.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GPU	Graphics Processing Unit	Grafična procesna enota
AES	Advanced Encryption Standard	Napredni standard za šifriranje
GPGPU	General Purpose Graphics Processing Unit	Grafična procesna enota za splošne namene
CUDA	Compute Unified Device Architecture	Enotna arhitektura za vzporedno računanje
OpenCL	Open Computing Language	Odprt jezik za vzporedno računanje na heterogenih sistemih
SIMD	Single Instruction Multiple Data	Ena procesna enota, različni tokovi podatkov
PTX	Parallel Thread Execution	Navidezni stroj za vzporedno izvajanje

Povzetek

V magistrskem delu smo se ukvarjali s primerjavo zaporednih implementacij bločnih šifer za CPE in njihovimi vzporednimi implementacijami za GPE. Pri tem smo si izbrali vseh pet finalistov standarda AES (Rijndael, Serpent, Twofish, MARS in RC6). Algoritme smo analizirali ter preučili možne izboljšave za njihovo vzporedno implementacijo. Implementirali smo vzporedne implementacije z razporejanjem podatkov ter merili pohitritev v primerjavi z zaporednimi implementacijami ter dosegli do dvajsetkratno pohitritev nekaterih algoritmov. Naredili smo tudi primerjavo med CUDO in OpenCL, platformama za pisanje večnitnih programov za grafične kartice. Implementirali smo tudi popolnoma svoje implementacije z bitnimi rezinami algoritmov Rijndael in Serpent za platformo CUDA ter ju primerjali z vzporednimi implementacijami z razporejanjem podatkov.

Ključne besede

vzporedni algoritmi, bločne šifre, kriptografija, CUDA, OpenCL, bitne rezine, AES, CTR, hitro šifriranje

Abstract

The aim of this Master's Thesis was to compare the serial implementations of block ciphers that run on CPU with corresponding parallel implementations that run on GPU. By analyzing the five finalists of the AES competition (Rijndael, Serpent, Twofish, MARS and RC6) we searched for possible improvements in their parallel implementations. Using the data parallelism techniques we implemented the algorithms in parallel and achieved the speed that was 20 times higher in comparison to the underlying serial implementations. We have also compared two different platforms for writing parallel programs on GPU: CUDA and OpenCL. In addition we implemented the bitslice implementations of algorithms Rijndael and Serpent for CUDA platform and compared them to data parallelism based implementations.

Keywords

parallel algorithms, block cyphers, cryptography, CUDA, OpenCL, bitslice, AES, CTR, fast encryption

Poglavje 1

Uvod

Kriptografski algoritmi so algoritmi, ki jih uporabljamo vsakodnevno, ne da bi se tega pravzaprav zavedali. Njihova naloga je, da neke zaupne podatke transformirajo na takšen način, da so nerazumljivi napadalcem. Uporabljajo se na veliko področjih. Od navadnega brskanja po spletu, branja elektronske pošte, šifriranja podatkov na trdem disku do avtentikacije uporabnikov in še marsikje drugje.

Pomemben dejavnik teh algoritmov je hitrost, saj lahko v nasprotnem primeru predstavlja ozko grlo v nekem sistemu. To je še posebej pomembno pri sistemih v realnem času, kjer si ne morem privoščiti predolgega čakanja za izvedbo neke zahteve. Eden izmed načinov, kako pohitriti izvedbo obstoječega algoritma je nakup močnejše strojne opreme. Težava pri tem je, da za nekajkratno pohitritev cena hitro preseže racionalne okvirje. Cenejši način je, da algoritem implementiramo vzporedno ter ga poženemo na več procesorjih ali jedrih hkrati. Pri tem se še posebej učinkovito izkažejo grafične kartice, ki vsebujejo veliko število jeder.

V magistrskem delu smo se osredotočili na bločne šifre, bolj podrobno na vseh pet finalistov izbora standarda AES (Rijndael, Serpent, Twofish, MARS, RC6). Algoritme smo natančno preučili ter ugotovili kako jih je mogoče najprej optimalno zaporedno implementirati v načinu CTR ter nato še vzporedno v načinu CTR na grafični kartici.

Zaporedne algoritme smo implementirali za centralno procesno enoto (CPE) v programskem jeziku C. Nato pa smo se lotili še vzporednih implementacij na osnovi razporejanja podatkov za grafične kartice (GPE) na platformah CUDA in OpenCL. Poleg tega smo preučili in implementirali tudi vzporedne implementacije z bitnimi rezinami (ang. bitslice) algoritmov Serpent in Rijndael na platformi CUDA. Ker do sedaj še ni bilo narejene takšne implementacije za platformo CUDA, smo morali razviti in implementirati čisto svoja algoritma.

Čase in prepustnosti zaporednih in vzporednih implementacij smo nato primerjali med seboj. Najprej smo primerjali zaporedne in vzporedne implementacije z razporejanjem podatkov, ki so bile implementirane na platformi CUDA. Nato smo naredili še primerjavo platform CUDA in OpenCL z istimi vzporednimi implementacijami. Nazadnje smo primerjali tudi vzporedne implementacije z razporejanjem podatkov in implementacije z bitnimi rezinami, ki smo jih implementirali na platformi CUDA.

Primerjave so pokazale, da smo dosegli do dvajsetkratno pohitritev glede na zaporedne implementacije. Najvišjo pohitritev je dosegel algoritem RC6. Primerjava platform CUDA in OpenCL je pokazala, da je prva hitrejša pri vseh petih algoritmih. Primerjava različnih načinov vzporednih implementacij pa je pokazala, da so bile implementacije z bitnimi rezinami počasnejše. Kljub temu menimo, da je bilo narejeno veliko, saj gre za prvo vrsto takšne implementacije bločne šifre na grafični kartici.

Zgradba magistrskega dela je sledeča: Drugo poglavje je namenjeno bločnim šifram, kjer so predstavljene osnove in načini delovanja bločnih šifer. V tretjem poglavju so predstavljeni finalisti izbora algoritma AES ter njihove podrobnosti. Četrto poglavje je namenjeno osnovam CUDE in OpenCL-ja, platformama za pisanje večnitnih programov na grafičnih karticah. Peto poglavje opisuje optimalne zaporedne implementacije na CPE ter šesto vzporedne na GPE. V sedmem poglavju so predstavljeni testni podatki, rezultati testiranja ter njihova razlaga. Zadnje poglavje je namenjeno sklepnim ugotovitvam in možnemu nadaljnjemu delu na to temo.

1.1 Sorodna dela

Ena izmed prvih implementacij finalista izbora AES za grafično kartico je bila implementacija Cooka in dr. [1] algoritma Rijndael s pomočjo OpenGL-ja. Dosežena prepustnost je bila 1,53 Mbps na grafični kartici Geforce3 Ti200. Podoben pristop so uporabili tudi Harrison in dr. [2], ki je implementiral Rijndael s pomočjo knjižnice DirectX9 in dosegel prepustnost okoli 870,8 Mbps na grafični kartici Geforce 7900GT. Težava, na katero so naleteli avtorji omenjenih člankov je bila, da te knjižnice niso podpirale logičnih in celoštevilskih operacij, ki so večinoma uporabljene v šifrirnih algoritmih. Posledično so dosegli tudi nižje prepustnosti.

S pojavitvijo GPGPU in platforme CUDA se je olajšalo tudi programiranje za grafične kartice. To je predstavljalo veliko revolucijo v računalniškem svetu, saj je nenadoma dalo izkoristiti veliko računsko moč, ki jo ponujajo grafične kartice. Posledično je bilo veliko računsko zahtevnih algoritmov uspešno predelanih za izvedbo na grafičnih karticah. Med njimi pa ne izostajajo niti bločne šifre.

Večji mejnik na tem področju je postavil Manavski [3], ki je z implementacijo algoritma Rijndael za platformo CUDA za grafično kartico Geforce 8800GTX dosegel prepustnost 8,28 Gbps. Avtor je uporabil štiri večje substitucijske tabele namesto ene manjše. Bloke podatkov je razdelil med štiri niti, podključe rund pa je shranil v deljenem pomnilniku. To delo je za nas pomembno, saj je avtor v njem pokazal, da je ključnega pomena kje se podatki hranijo in kakšne tabele se uporabljajo.

Štiri večje substitucijske tabele so uporabili tudi Harrison in dr. [4], ki so raziskovali tudi katera vrsta pomnilnika je najbolj optimalna za hranjenje tabel. Pri tem so prišli do ugotovitve, da hranjenje tabel v deljenem pomnilniku močno pohitri delovanje algoritma.

Implementacijo algoritma Rijndael v načinu CTR za platformo CUDA so kasneje pohitrili še Di Biagio in dr. [5]. Avtorji so primerjali fino delitev podatkov, pri kateri imamo štiri niti na blok podatkov in grobo delitev podatkov, kjer vsaka nit skrbi za svoj blok podatkov. Primerjali so tudi načine

hranjenja substitucijskih tabel v konstantnem in deljenim pomnilnikom. Poleg tega so preverili tudi kolikšna velikost bloka niti je najbolj optimalna. Prišli so do zaključka, da je najhitrejši način groba delitev podatkov, s hranjenjem substitucijskih tabel v deljenem pomnilniku in blokom niti velikosti 256. Pri tem so dosegli prepustnost 12,4 Gbps na kartici Geforce 8800GT.

S podobnimi testi so se ukvarjali tudi Iwai in dr. [6], ki so ugotovili, da je najbolj optimalna razdelitev ena nit na blok podatkov. Hranjenje čistopisa in šifropisa predlagajo v globalnem pomnilniku, substitucijske tabele in ključe pa v konstantnem. Dosegli so prepustnost 35,2 Gbps na Geforce GTX 285. Avtorji poleg tega opozarjajo tudi na pasti pri implementaciji kot so zaporedni dostop do podatkov itd. Mei in dr. [7] so uporabili drugačno delitev s 16 niti na blok podatkov. Poleg tega obravnavajo različne načine prenosa in hrambe podatkov za njihov način delitve. Dosegli so prepustnost 6,4 Gbps na kartici Geforce 9200M GS.

Li in dr. [8] pa so z delitvijo ene niti na blok podatkov in hranjenjem tabel v deljenem pomnilniku dosegli prepustnost okoli 60 Gbps na kartici Tesla C2050. Poleg tega v članku opozorijo, kako doseči zaporedni dostop do globalnega pomnilnika.

Izmed implementacij ostalih algoritmov na grafičnih karticah so večjo primerjavo naredili Nishikawa [9] in dr., ki so testirali in primerjali algoritme AES (Rijndael), Camellia, CIPHERUNICORN-A in Hierocrypt-3 na platformi CUDA. Njihove optimalne implementacije ponovno uporabljajo eno nit na blok podatkov in hranjenje substitucijskih tabel in ključev v deljenem pomnilniku.

Primerjavo med implementacijami finalistov Rijndael, Serpent in Twofish za platformo OpenCL so naredili Wang [10] in dr. Avtorji poročajo, da povprečno dosegajo 10 do 20 procentov manjšo prepustnost kot podobne implementacije na CUDI drugih avtorjev. Al Shamsi in Al Ali [11] sta implementirala in primerjala iste algoritme na platformi CUDA. Nazlee in dr. [12] pa so implementirali Serpent za CUDO. Pri tem pa so podobno, kot nekateri avtorji algoritma Rijndael za CUDO, uporabili eno nit na podatkovni blok.

Pri pregledu naštetih del smo prišli do ugotovitve, da je ključni del uspešne implementacije ustrezna razdelitev podatkov na niti in pametna uporaba hierarhije pomnilnikov, ki so nam na voljo na grafični kartici.

Implementacije bločnih šifer z bitnimi rezinami na grafični kartici so dokaj novo področje, saj do sedaj ni bilo narejeno še nobene primerjave z ostalimi implementacijami. Prvo implementacijo z bitnimi rezinami je predstavil Biham [13], ki je implementiral DES z bitnimi rezinami za CPE. Reberio in dr. [14] pa Rijndael z bitnimi rezinami za CPE. Predvsem slednje delo nam je koristilo kot zgled pri implementaciji algoritma Rijndael z bitnimi rezinami za platformo CUDA.

Naše delo se od ostalih razlikuje v tem, da smo primerjali zaporedne in vzporedne implementacije vseh petih finalistov izbora algoritma AES, saj lahko zaradi različne zgradbe algoritmov pričakujemo različne pohitritve. Naredili smo primerjavo vzporednih implementacij na platformi CUDA in platformi OpenCL, ker smo želeli preizkusiti ali sta platformi primerljivi pri našem problemu. Poleg standardnih vzporednih implementacij z razporejanjem podatkov smo implementirali in primerjali tudi čisto svoje vzporedne implementacije z bitnimi rezinami algoritmov Rijndael in Serpent za platformo CUDA.

O implementaciji z bitnimi rezinami algoritma Rijndael za grafični procesor in njeni primerjavi z zaporedno implementacijo ter implementacijo z razporejanjem podatkov pa smo napisali tudi članek z naslovom “Parallel Bitslice AES-CTR implementation on CUDA“, ki smo ga poslali v objavo v revijo Journal of Parallel and Distributed Computing [29].

Poglavje 2

Bločne šifre

Definicija 2.0.1 *Kriptosistem je peterica* $(\mathcal{P}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$*, kjer velja:*

1. $\mathcal{P}$ *je končna množica čistopisov,*
2. $\mathcal{C}$ *je končna množica šifropisov,*
3. $\mathcal{K}$ *je končna množica ključev.*
4. $\mathcal{E} = \{E_k | k \in \mathcal{K}\}$ *je množica šifrirnih funkcij* $E_k : \mathcal{P} \to \mathcal{C}$*.*
5. $\mathcal{D} = \{D_k | k \in \mathcal{K}\}$ *je množica dešifrirnih funkcij* $D_k : \mathcal{C} \to \mathcal{P}$*.*
6. *Za vsak ključ* $e \in \mathcal{K}$*, obstaja ključ* $d \in \mathcal{K}$*, tako da za vsak* $p \in \mathcal{P}$ *velja:*

$$D_d(E_e(p)) = p.$$

Če je $e = d$*, je kriptosistem simetričen, drugače je asimetričen.*

Simetrične kriptosisteme nadalje delimo na dve vrsti: tokovne in bločne šifre. Za tokovne šifre je značilno, da najprej ustvarijo tok ključev $z = z_1 z_2 \ldots$, ki ga nato uporabijo za šifriranje posameznih bitov ali znakov:

$$y = y_1 y_2 \ldots = E_{z_1}(x_1) E_{z_2}(x_2) \ldots .$$

Bločne šifre pa po drugi strani vzamejo blok podatkov fiksne dolžine n in ključ fiksne dolžine k. Blok obdelajo kot celoto ter nato vrnejo izhod dolžine n. V našem magistrskem delu se bomo ukvarjali s pohitritvijo bločnih šifer.

Vse bločne šifre, ki jih bomo implementirali mi, so t.i. iteracijsko-produktne šifre. Za njih je značilno, da imajo definiran algoritem za razširjanje ključa ter funkcijo runde, ki jo uporabijo N_r-krat zapored, kjer N_r predstavlja število rund.

Njihovo delovanje je sledeče: Imejmo vhodni blok x in ključ K. Najprej iz ključa K, z algoritmom za razširjanje ključa, izračunamo podključe $(K^1, K^2, \ldots, K^{Nr})$. V vsaki rundi r nato funkcija runde g vzame prejšnje stanje w^{r-1} in trenuten podključ K^r ter izračuna novo stanje $w^r = g(w^{r-1}, K^r)$. Začetno stanje w^0 je čistopis, končno stanje po Nr rundah pa šifropis.

Celoten postopek je torej:

$$\begin{aligned} w^0 &\leftarrow x \\ w^1 &\leftarrow g(w^0, K^1) \\ w^2 &\leftarrow g(w^1, K^2) \\ &\vdots \\ w^{Nr} &\leftarrow g(w^{Nr-1}, K^{Nr}) \\ y &\leftarrow w^{Nr} \end{aligned}$$

Dešifriranje poteka v obratnem vrstnem redu. Pri tem pa moramo uporabiti inverzno funkcijo $w^r = g^{-1}(w^{r+1}, K^r)$, zato mora biti g injektivna pri fiksnem K^r. Velja torej: $g^{-1}(g(w, K^r), K^r) = w$. [15]

2.1 Načini delovanja

Ker šifrirajo bločne šifre po en blok dolžine n, se postavi vprašanje, kaj narediti takrat, kadar imamo podatke daljše kot n. Za ta namen imamo različne načine delovanja (ang. Modes of Operations), ki omogočajo šifriranje podatkov poljubne dolžine. Nekatere najbolj znani načini delovanja so:

- Način elektronske kodne knjige (ang. Electronic Codebook Mode, kratica: ECB),
- Način veriženja šifriranih blokov (ang. Cipher Block Chaining Mode, kratica: CBC),
- Način odziva izhoda (ang. Output Feedback Mode, kratica: OFB),
- Način odziva šifropisa (ang. Cipher Feedback Mode, kratica: CFB),
- Način štetja (ang. Counter Mode, kratica: CTR)

2.1.1 Način ECB

To je najbolj osnoven način, kjer zaporedje blokov $x_1,\ x_2,\ \ldots,\ x_n$ neodvisno šifriramo z istim ključem K, da dobimo šifrirano zaporedje blokov $y_1,\ y_2,\ \ldots,\ y_n$. Slabost tega načina je, da je šifropis dveh enakih blokov ($x_i = x_j$) prav tako enak ($y_i = y_j$). Posledica tega je, da je iz šifropisa možno razbrati določene vzorce, ki veljajo tudi za čistopis. Zaradi tega se uporabo tega načina odsvetuje.

2.1.2 Način CBC

Pri tem načinu se med biti šifropisa prejšnjega bloka y_{i-1} in biti trenutnega čistopisa x_i izvede operacija XOR. Začnemo z začetnim vektorjem (ang. Initialization Vector)

$$y_0 = IV$$

in nadaljujemo po formuli

$$y_i = E_K(y_{i-1} \oplus x_i),$$

da dobimo šifropis $y_1,\ y_2,\ \ldots,\ y_n$. Za dešifriranje uporabimo obratno formulo

$$x_i = y_{i-1} \oplus D_K(y_i).$$

Slabost tega načina je, da je šifriranje nemogoče implementirati vzporedno.

2.1.3 Način OFB

Z načinom OFB dejansko ustvarimo sinhrono tokovno šifro iz bločne šifre. Deluje tako, da izpeljemo tok ključev $z_1, z_2, \ldots, z_n$ iz začetnega vektorja. To naredimo tako, da določimo

$$z_0 = IV.$$

Naslednji elementi toka so izračunani po formuli

$$z_i = E_K(z_{i-1}).$$

Zaporedje blokov se nato šifrira po pravilu

$$y_i = x_i \oplus z_i$$

ter dešifrira po pravilu

$$x_i = y_i \oplus z_i.$$

V obeh primerih se uporabi samo šifrirna funkcija E_K.

2.1.4 Način CFB

Ta način deluje podobno kot OFB. Razlikuje se v tem, kako naredimo tok ključev $z_1,\ z_2,\ \ldots,\ z_n$. Tega naredimo tako, da najprej določimo $y_0 = IV$.

Elementi toka pa se izračunajo po formuli:

$$z_i = E_K(y_{i-1}).$$

Bloke se nato zašifrira in dešifrira po enaki formuli kot pri načinu OFB.

2.1.5 Način CTR

CTR je podoben prejšnjima dvema načinoma. Razlika je v tem, da se da tok ključev ustvariti vzporedno.

Deluje tako, da najprej naključno izberemo neko vrednost števca *ctr* v $\{0,1\}^n$. Nato ustvarimo m blokov dolžine n po formuli:

$$T_i = \texttt{ctr} + (i-1) \mod 2^n, \ (i = 1, \ldots, m).$$

Šifriranje podatkov nato poteka po formuli:

$$y_i = x_i \oplus E_K(T_i).$$

Pri tem načinu je seveda pomembno, da izberemo nov `ctr`, preden se vrednost števca obrne naokoli. To pomeni, da ne smemo imeti dveh vrednosti T_i in T_j, za kateri velja $T_i = T_j, i \neq j$.

Poglavje 3

AES

Leta 1997 je ameriški inštitut za standarde in tehnologijo (NIST) začel postopek izbire standardnega algoritma na področju simetrične bločne kriptografije (Advanced Encryption Standard, AES), ki bi zamenjal uveljavljen standard DES. Glavni razlog za menjavo je bila ranljivost na napad z izčrpnim iskanjem (ang. Exhaustive Search Attack), predvsem zaradi kratke dolžine ključa (56 bitov) [30].

Zahteve standarda so bile, da uporablja bloke dolžine 128 bitov in da podpira dolžino ključev 128, 192, in 256 bitov. Poleg tega je bila zahteva tudi, da naj bo algoritem odprt.

Izmed enaindvajset prijavljenih algoritmov jih je petnajst ustrezalo zahtevam, med katerimi pa se je petim uspelo uvrstiti v finale. Ti algoritmi so bili Rijndael, Serpent, Twofish, MARS in RC6.

Na koncu je novi standard AES postal algoritem Rijndael. Razlog za to je bila boljša kombinacija varnosti, hitrosti in fleksibilnosti od ostalih algoritmov [15].

V nadaljevanju poglavja so opisani algoritmi po vrstnem redu finalnega izbora. Pri opisu algoritmov pomeni pojem beseda 32-bitni podatek, oziroma štirje zaporedni bajti zapisani po pravilu tankega konca.

3.1 Rijndael

Rijndael je substitucijsko-linearno transformacijsko omrežje, kar pomeni, da je funkcija runde sestavljena iz substitucijskih tabel in linearnih transformacij. Algoritem deluje v 10, 12 ali 14 rundah, odvisno od dolžine ključa. Rijndael najpreprostejše opiše psevdokoda 3.1. Funkcija `Round` pa je definirana v psevdokodi 3.2.

```
1 RijndaelEncrypt(State, CipherKey){
2     KeyExpansion(CipherKey, ExpandedKey);
3     AddRoundKey(State, ExpandedKey);
4     for(i = 1; i < Nr; i++){
5         EncryptRound(State, ExpandedKey+4*i, false);
6     }
7     EncryptRound(State, ExpandedKey+4*Nr, true);
8 }
```

Psevdokoda 3.1: Psevdokoda algoritma Rijndael.

```
1 EncryptRound(State, RoundKey, boolean LastRound){
2     SubBytes(State);
3     ShiftRows(State);
4     if(!LastRound){
5         MixColumns(State);
6     }
7     AddRoundKey(State, RoundKey);
8 }
```

Psevdokoda 3.2: Funkcija runde algoritma Rijndael.

3.1.1 Stanje

Stanje algoritma Rijndael (ang. State) si lahko predstavljamo kot matriko bajtov velikosti 4 krat 4. Začetno stanje dobimo tako, da vhodne podatke po bajtih $b_0,\ b_1,\ \ldots,\ b_{15}$ zložimo v stolpce.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \leftarrow \begin{pmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{pmatrix}$$

Isti način uporabimo tudi na koncu, da iz stanja dobimo šifrirane podatke.

3.1.2 SubBytes

Operacija `SubBytes` je nelinearna preslikava, ki jo izvedemo nad vsakim bajtom stanja. Izvedemo jo s pomočjo vnaprej izračunane substitucijske tabele (`S-box`). Elementi tabele so izračunani kot inverzni element operacije množenja v končnem obsegu, čemur sledi še afina transformacija. Postopek za izračun je sledeč.

Najprej preslikamo bajt b v polinom v končnem obsegu. To naredimo tako, da bajt b, ki ga sestavljajo biti $b_7,\ b_6,\ \ldots,\ b_0$ preslikamo v polinom sedme stopnje po pravilu:

$$b(x) = \sum_{i=0}^{7} b_i x^i.$$

Nato poiščemo njegov inverzni element operacije množenja $b^{-1}(x)$ v končnem obsegu:

$$GF(2^8) = GF(2)[x]/(x^8 + x^4 + x^3 + x + 1),$$

tako da velja:

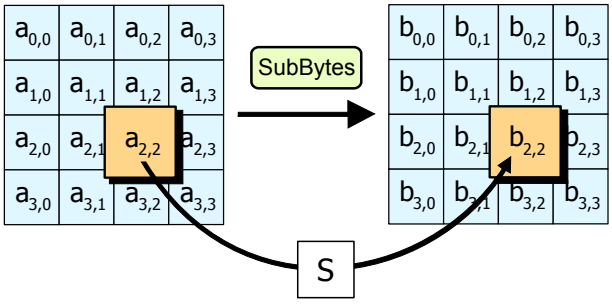
$$b(x)b^{-1}(x) \equiv 1 \pmod{x^8 + x^4 + x^3 + x + 1}$$

Robni primer je 0, ki pa se preslika vase.

Končni obseg je obseg z omejenim številom elementov. En način predstavitve končnega obsega so cela števila z operacijama seštevanja in množenja po modulu velikosti množice. Najbolj preprost primer je $GF(2) = \{0, 1\}$ z dvema elementoma ter operacijama seštevanja in množenja po modulu 2. Drug način predstavitve pa so polinomi z elementi v nekem drugem končnem obsegu:

$$GF(q) = GF(p)[x]/(P),$$

kjer je $q = p^n$, P pa nedeljiv polinom stopnje n, ki ni produkt polino-



Slika 3.1: Operacija `SubBytes` nad enim elementom stanja. Vir: Prirejeno po [33].

mov znotraj $GF(q)$. V tem primeru imamo seštevanje definirano kot XOR operacijo istoležnih koeficientov polinoma in množenje kot množenje polinomov po modulu nedeljivega polinoma P. Preprost primer je $GF(2^2) = GF(2)[x]/(x^2 + x + 1) = \{0, 1, x, x + 1\}$.

Ko imamo izračunan inverzni element operacije množenja, nad njim izvedemo še afino transformacijo, ki je definirana s formulo:

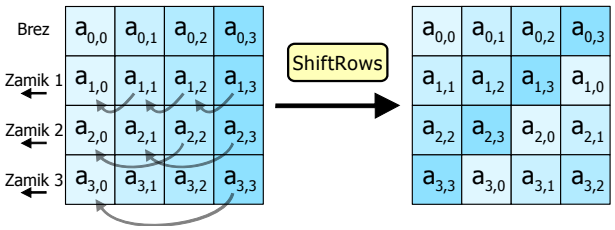
$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}.$$

Kjer $x_0, \ldots, x_7$ predstavljajo bite izračunanega inverza.

V večini implementacij algoritma Rijndael imamo `SubBytes` implementiran kot poizvedbo v tabelo, saj se s tem izognemo zamudnemu računanju inverznega elementa.

3.1.3 ShiftRows

Operacija `ShiftRows` rotira vrstice stanja za določeno število. Prva vrstica se pusti pri miru. Druga se zamakne za en bajt v levo, tretja za dva in četrta za tri, kot to prikazuje slika 3.2



Slika 3.2: Operacija `ShiftRows` nad vrsticami stanja. Vir: Prirejeno po [33].

3.1.4 MixColumns

`MixColumns` je operacija, ki jo izvedemo tako, da vsak stolpec stanja modularno pomnožimo s polinomom tretje stopnje $c(x) = 3x^3 + x^2 + x + 2$, ki ima koeficiente v istem končnem obsegu $GF(2^8)$ kot pri operaciji `SubBytes`.

Najprej stolpec, ki ga sestavljajo bajti a_3, a_2, a_1 in a_0, preslikamo v polinom tretje stopnje s koeficienti v $GF(2^8)$:

$$a(x) = a_0 + a_1x + a_2x^2 + a_3x^3.$$

Produkt dveh polinomov tretje stopnje je polinom šeste stopnje:

$$\begin{aligned} d(x) &= a(x)b(x) \\ &= (a_0 + a_1x + a_2x^2 + a_3x^3)(b_0 + b_1x + b_2x^2 + b_3x^3) \\ &= a_0b_0 \\ &+ (a_1b_0 + a_0b_1)x \\ &+ (a_2b_0 + a_1b_1 + a_0b_2)x^2 \\ &+ (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)x^3 \\ &+ (a_3b_1 + a_2b_2 + a_1b_3)x^4 \\ &+ (a_3b_2 + a_2b_3)x^5 \\ &+ a_3b_3x^6. \end{aligned}$$

Tega ne moremo zapisati kot vektor štirih bajtov, zato ga delimo s polinomom četrte stopnje in vzamemo njegov ostanek. V Rijndaelu je uporabljen polinom $M(x) = x^4 + 1$, ki ima to lastnost, da velja

$$x^i \equiv x^{i \mod 4} \pmod{x^4 + 1} \text{ [31]}.$$

Polinom $d(x)$ postane v tem primeru:

$$\begin{aligned} d(x) &= d_0 + d_1x + d_2x^2 + d_3x^3 \\ &= (a_0b_0 + a_3b_1 + a_2b_2 + a_1b_3) \\ &+ (a_1b_0 + a_0b_1 + a_3b_2 + a_2b_3)x \\ &+ (a_2b_0 + a_1b_1 + a_0b_2 + a_3b_3)x^2 \\ &+ (a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3)x^3 \end{aligned}$$

Koeficiente produkta $d(x)$ lahko izrazimo na sledeči način:

$$\begin{aligned} d_0 &= a_0b_0 + a_3b_1 + a_2b_2 + a_1b_3 \\ d_1 &= a_1b_0 + a_0b_1 + a_3b_2 + a_2b_3 \\ d_2 &= a_2b_0 + a_1b_1 + a_0b_2 + a_3b_3 \\ d_3 &= a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3. \end{aligned}$$

Kar pa lahko zapišemo tudi kot matrično množenje:

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

V primeru polinoma $c(x) = 3x^3 + x^2 + x + 2$ je matrika enaka:

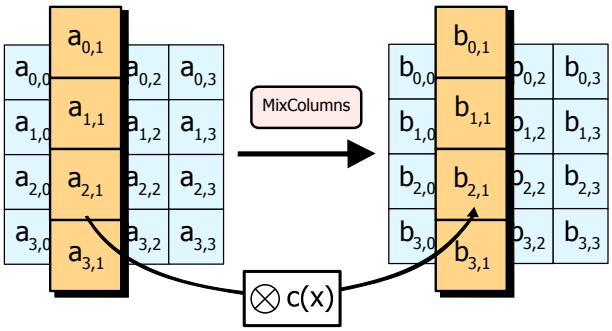
$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

Za dešifriranje pa uporabimo drug polinom $d(x)$:

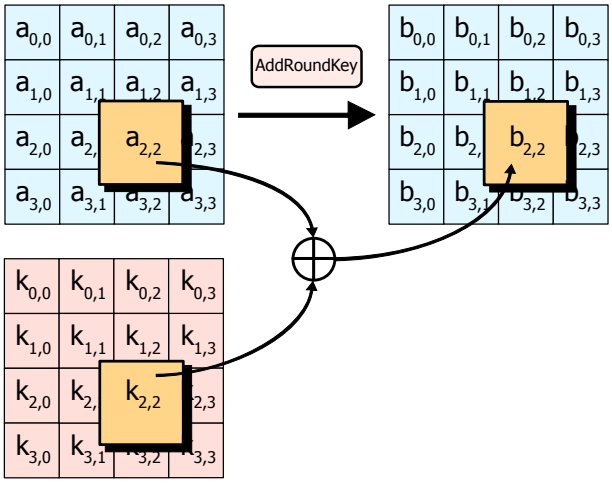
$$d(x) = 11x^3 + 13x^2 + 9x + 13$$

3.1.5 AddRoundKey

Delovanje `AddRoundKey` je preprosto, saj gre za XOR operacijo med biti stanja in biti podključa trenutne runde, ki je prav tako vektor 16 bajtov oziroma 128 bitov. Delovanje prikazuje slika 3.4



Slika 3.3: Operacija `MixColumns` nad stolpcem stanja. Vir: Prirejeno po [33].



Slika 3.4: Operacija `AddRoundKey` nad elementom stanja. Vir: Prirejeno po [33].

3.1.6 Razširjanje ključa

Razširjanje ključa `KeyExpansion` je funkcija, ki sprejme ključ in izračuna podključe rund ali razširjen ključ `ExpandedKey`. Psevdokoda za funkcije za izračun podključev 3.3 sprejme seznam bajtov ključa, ki je dolžine $4N_k$. N_k nam pove iz koliko štiri bajtnih besed je sestavljen ključ: 4, 6 ali 8. Poleg tega sprejme tudi seznam štiri bajtnih besed dolžine $4 * (N_r + 1)$ za shranjevanje izhoda podključev vsake runde.

```
1  KeyExpansion(byte Key[4*Nk], word W[4*(Nr+1)])
2  {
3      for(i = 0; i < Nk, i++)
4          W[i] = (Key[4*i], Key[4*i+1], Key[4*i+2], Key[4*i+3])
5      for(i = Nk; i < 4 * (Nr + 1); i++){
6          temp = W[i-1];
7          if( i % Nk == 0){
8              temp = SubWord(RotWord(temp))^ Rcon[i/Nk];
9          }
10         else if(Nk > 6 and i % Nk == 4){
11             temp = SubWord(temp)
12         }
13         W[i] = W[i - Nk] ^ temp;
14     }
15 }
```

Psevdokoda 3.3: Razširjanje ključa algoritma Rijndael.

Funkcija `SubWord(W)` izračuna `SubBytes` nad posameznim bajtom besede W. Funkcija `RotWord(W)` je rotacija besede W za en bajt v levo: $(a, b, c, d) \rightarrow (b, c, d, a)$. `Rcon` je konstantna tabela, katere elementi so definirani kot $Rcon[i] = (RC[i], 0, 0, 0)$, kjer je $RC[i]$ tabela elementov v $GF(2^8)$, ki se izračunajo po pravilu:

$$RC[1] = 1$$
$$RC[i] = xRC[i-1] = x^{(i-1)} \pmod{x^8 + x^4 + x^3 + x + 1}\ [31].$$

Iz psevdokode je razvidno, da je prvih Nk besed enakih tistim iz ključa. Nadaljnje vrednosti pa so izračunane iterativno iz vrednosti prejšnjih podključev.

3.2 Serpent

Serpent je substitucijsko-linearno transformacijsko omrežje z 32 rundami. Njegova posebnost je, da je bil načrtovan tako, da omogoča enostavno implementacijo substitucijskih tabel samo z bitnimi operacijami (AND, OR, XOR, NOT), zamiki in prirejanji.

Funkcija runde sestoji iz treh plasti:

1. operacije XOR s podključem runde,
2. 32 preslikav z ustrezno $\{0,1\}^4 \rightarrow \{0,1\}^4$ tabelo in
3. linearne transformacije.

V zadnji rundi se linearna transformacija nadomesti z operacijo XOR z zadnjim podključem.

Stanje algoritma je sestavljeno iz štirih besed. Algoritem najboljše opiše psevdokoda 3.4. Funkcija `EncryptRound` pa je prikazana v psevdokodi 3.5.

```
1 SerpentEncrypt(State,CipherKey){
2     KeyExpansion(CipherKey, ExpandedKey);
3     InitialPermutation(State);
4     for(i = 0; i < 33; i++)
5         EncryptRound(State, ExpandedKey, i);
6
7     FinalPermutation(State);
8 }
```

Psevdokoda 3.4: Psevdokoda algoritma Serpent.

```
1 EncryptRound(State, ExpandedKey, i){
2     KeyMixing(State, ExpandedKey+4*i);
3     SBox(State, i % 8);
4     if(i < 31)
5         LinearTransform(State);
6     else
7         KeyMixing(State, ExpandedKey+4*32);
8 }
```

Psevdokoda 3.5: Psevdokoda runde algoritma Serpent.

Namen funkcij `InitialPermutation` in `FinalPermutation` je, da preoblikujemo podatke v drugo obliko, kadar hočemo uporabiti preračunane substitucijske tabele.

3.2.1 Substitucijske tabele

Pri algoritmu Serpent uporabljamo 8 tabel z indeksi od 0 do 7. Izbiramo jih po ključu `i mod 8`, kjer `i` predstavlja številko runde. Vsako tabelo torej uporabimo štirikrat v 32 rundah. Tabele preslikajo štiri bite v štiri bite.

Kot smo že omenili, je njihova posebnost to, da jih lahko implementiramo kot zaporedje preprostih bitnih operacij, zamiki in prirejanji med štirimi besedami stanja. Ta način je učinkovitejši, saj se z njim izognemo 32 poizvedbam znotraj ene runde. [16]

3.2.2 LinearTransform

Delovanje funkcije `LinearTransform` prikazuje slika 3.5. Na sliki predstavljajo X_0, X_1, X_2 in X_3 vhodne besede stanja. Znak $<<$ predstavlja zamik bitov besede v levo in znak $<<<$ rotacijo bitov besede. Razlika med zamikom in rotacijo je v tem, da se pri zamiku na skrajni desni konec zapišejo ničle, pri rotaciji pa se zapišejo biti, ki izpadejo pri zamiku levo.

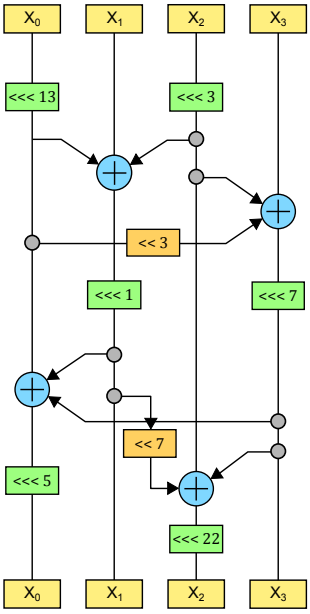
3.2.3 KeyMixing

`KeyMixing` je podobna funkcija kot `AddRoundKey` pri algoritmu Rijndael. Rezultat funkcije je XOR operacija med štirimi besedami stanja in podključem trenutne runde.

3.2.4 Razširjanje ključa

Za delovanje algoritma Serpent potrebujemo 132 besed podključev.

Najprej ključ dopolnimo do polne velikosti, če je njegova dolžina krajša od 256 bitov. To naredimo tako, da pripnemo bajt z vrednostjo 1 na konec



Slika 3.5: Operacija `LinearTransform` nad stanjem v algoritmu Serpent. Vir: Prirejeno po: [34].

ključa ter dodamo še toliko ničel, da je dolžina enaka 256.

Nato razširimo ključ na 33 128-bitnih podključev $K_0, \ldots, K_{32}$. Najprej zapišemo ključ kot osem besed $w_{-8}, \ldots, w_{-1}$ in jih razširimo v vmesni ključ $w_0, \ldots, w_{131}$, ki mu rečemo predključ. To naredimo po naslednji rekurzivni enačbi:

$$w_i = (w_{i-8} \oplus w_{i-5} \oplus w_{i-3} \oplus w_{i-1} \oplus \phi \oplus i) <<< 11$$

Simbol ϕ v enačbi predstavlja decimalni del zlatega reza $(\sqrt{5} + 1)/2$ ali 0x9e3779b9 v šestnajstiškem zapisu.

Podključi rund so izračunani iz predključa s pomočjo substitucijskih tabel

$S_0, S_1, \ldots S_7$ na naslednji način:

$$\begin{aligned}
\{k_0, k_1, k_2, k_3\} &= S_3(w_0, w_1, w_2, w_3) \\
\{k_4, k_5, k_6, k_7\} &= S_2(w_4, w_5, w_6, w_7) \\
\{k_8, k_9, k_{10}, k_{11}\} &= S_1(w_8, w_9, w_{10}, w_{11}) \\
\{k_{12}, k_{13}, k_{14}, k_{15}\} &= S_0(w_{12}, w_{13}, w_{14}, w_{15}) \\
\{k_{16}, k_{17}, k_{18}, k_{19}\} &= S_7(w_{16}, w_{17}, w_{18}, w_{19}) \\
&\vdots \\
\{k_{124}, k_{125}, k_{126}, k_{127}\} &= S_4(w_{124}, w_{125}, w_{126}, w_{127}) \\
\{k_{128}, k_{129}, k_{130}, k_{131}\} &= S_3(w_{128}, w_{129}, w_{130}, w_{131})
\end{aligned}$$

Nato sestavimo besede k_j v 128-bitne podključe K_i:

$$K_i = \{k_{4i}, k_{4i+1}, k_{4i+2}, k_{4i+3}\}$$

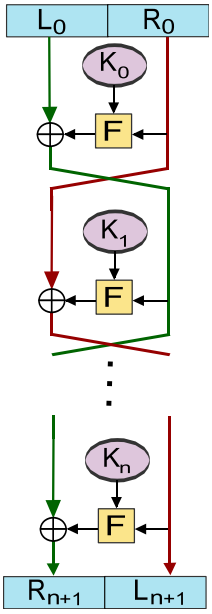
Kadar uporabljamo preračunane tabele, moramo nad posameznim podključem uporabiti še funkcijo `InitialPermutation`.

3.3 Twofish

Twofish je Feistelovo omrežje s 16 rundami in dodatnim beljenjem (ang. whitening) na začetku in koncu. Njegova posebnost je, da uporablja substitucijske tabele, ki so odvisne od vhodnega ključa. Algoritem najbolje razloži njegova shema na sliki 3.7. Stanje pa je sestavljeno iz štirih besed.

Feistelovo omrežje je kriptografska metoda, ki spremeni neko funkcijo F v permutacijo. Uporabljena je v številnih algoritmih, tudi v DES. Glavni gradnik Feistelovega omrežja je funkcija F, ki mora biti vedno nelinearna in po možnosti tudi ne surjektivna. Njena definicija je sledeča:

$$F : \{0,1\}^{n/2} \times \{0,1\}^N \mapsto \{0,1\}^{n/2},$$

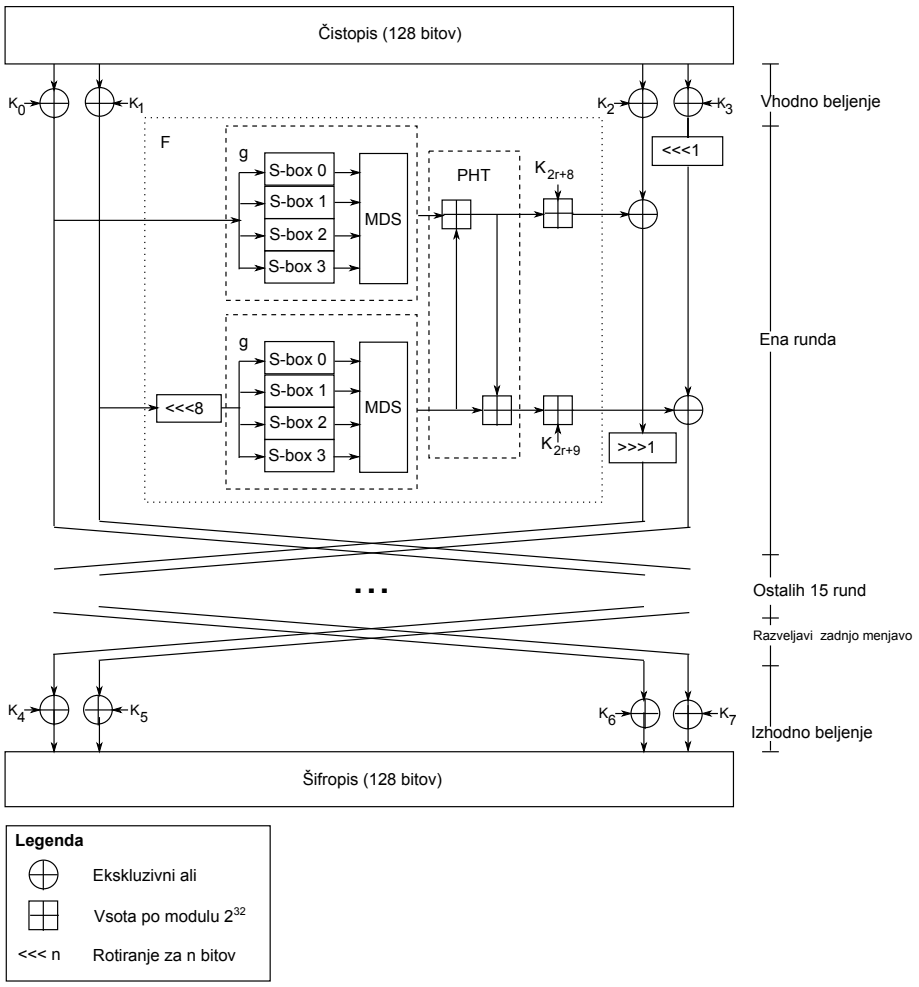


Slika 3.6: Feistelovo omrežje. Vir: Prirejeno po: [35]

kjer n predstavlja dolžino bloka. F torej vzame $n/2$ bitov bloka in ključ dolžine N ter vrača izhod dolžine $n/2$. Omrežje deluje tako, da se vhod v omrežje razdeli na dva bloka. Izvornega in ciljnega. Izvorni blok se pošlje, skupaj s ključem v funkcijo F, katere izhod se nato sešteje po modulu dva s ciljnim blokom. Bloka se nato zamenjata. Dvakratno ponovitev te operacije imenujemo cikel Feistelovega omrežja. Delovanje Feistelovega omrežja prikazuje slika 3.6.

3.3.1 Beljenje

Beljenje (ang. whitening) je, podobno kot pri Rijndaelu in Serpentu, XOR operacija med stanjem in ustreznimi podključi. Uporaba beljenja drastično pomaga pri preprečevanju napadov s skrajšanjem algoritma. [17]



Slika 3.7: Zgradba algoritma Twofish. Vir: Prirejeno po: [36]

3.3.2 Funkcija F

Funkcija F v algoritmu Twofish je 64-bitna permutacija. Sprejme tri argumente. Dve vhodni besedi R_0 in R_1 in številko runde r za izbiro ustreznega podključa.

Deluje tako, da najprej skozi funkcijo g pošljemo argument R_0, da dobimo T_0. Nato rotiramo še R_1 v levo za en bajt ter jo pošljemo skozi g, da dobimo T_1. T_0 in T_1 nato uporabimo v PHT. PHT ali Psevdo-Hadamardova transformacija je operacija, ki je definirana kot:

$$\begin{aligned} a' &= a + b \mod 2^{32} \\ b' &= a + 2b \mod 2^{32} \end{aligned}$$

Implementacija na 32-bitnih procesorjih je enostavna, saj gre za navadno seštevanje.

Z enačbami lahko funkcijo F zapišemo kot:

$$\begin{aligned} T_0 &= g(R_0) \\ T_1 &= g(R_1 <<< 8) \\ F_0 &= (T_0 + T_1 + K_{2r+8}) \mod 2^{32} \\ F_1 &= (T_0 + 2T_1 + K_{2r+9}) \mod 2^{32} \end{aligned}$$

3.3.2.1 Funkcija g

Vhodna 32-bitna beseda X se razdeli na štiri bajte (x_0, x_1, x_2, x_3) po pravilu tankega konca. Te nato pošljemo vsakega skozi svojo substitucijsko tabelo. Tabele so bijektivne 8×8 permutacije in so pogojene z vhodnim ključem.

Izhodi tabel se nato obravnavajo kot vektor $(y_0,\ y_1,\ y_2,\ y_3)$ z elementi v končnem obsegu

$$GF(2^8) = GF(2)[x]/(x^8 + x^6 + x^5 + x^3 + 1),$$

ki ga pomnožimo z matriko MDS:

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Vektor bajtov $(z_0,\ z_1,\ z_2,\ z_3)$ nato sestavimo nazaj kot besedo Z po pravilu tankega konca.

3.3.3 Razširjanje ključa in računanje substitucijskih tabel

Za algoritem Twofish moramo izračunati 40 besed podključev $K_0,\ \ldots,\ K_{39}$. Poleg podključev nam vhodni ključ definira tudi štiri tabele.

Najprej definirajmo $k = N/64$, kjer je N dolžina ključa v bitih. Vhodni ključ M je sestavljen iz $8k$ bajtov $m_0, \ldots, m_{8k-1}$ oziroma $2k$ besed M_i.

Besede zložimo v sodi in lihi vektor dolžine k:

$$M_e = (M_0, M_2, \ldots, M_{2k-2})$$
$$M_o = (M_1, M_3, \ldots, M_{2k-1})$$

Za izpeljavo substitucijskih tabel potrebujemo še tretji vektor S. Dobimo ga tako, da naredimo vektorje po osem bajtov ključa M in interpretiramo bajte kot elemente v

$$GF(2^8) = GF(2)[x]/(x^8 + x^6 + x^3 + x^2 + 1).$$

Pri tem se uporabi drugačen nedeljivi polinom kot pri množenju z matriko MDS. Vektorje nato pomnožimo z matriko RS, velikosti 4×8, ki je defini-

rana v specifikaciji algoritma [17].

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} . & \cdots & . \\ \vdots & RS & \vdots \\ . & \cdots & . \end{pmatrix} \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

Rezultat množenja je vektor velikosti štirih bajtov, ki ga, po pravilu tankega konca, zložimo v besede S_i.

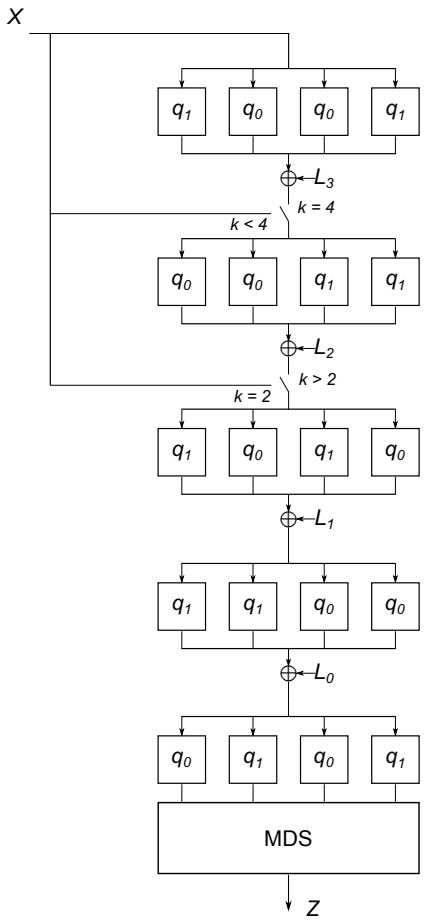
Besede nato v obratnem vrstnem redu zložimo v vektor dolžine k:

$$S = (S_{k-1}, S_{k-2}, \ldots, S_0)$$

3.3.3.1 Funkcija h

Za izpeljavo podključev in substitucijskih tabel moramo najprej definirati funkcijo h. Funkcija h sprejme besedo X in seznam besed $L = (L_0, \ldots, L_{k-1})$. Njeno shemo prikazuje slika 3.8.

Funkcija deluje tako, da vhodno besedo X najprej razdeli na štiri bajte po pravilu tankega konca in jih pelje čez kaskado tabel in operacij XOR z besedami iz seznama L. Čez koliko faz bodo šli bajti, je odvisno od dolžine seznama L oziroma vhodnega ključa. Tabeli q_0 in q_1 sta fiksni permutaciji 8×8. Operacija MDS je ista kot pri funkciji g. [17]



Slika 3.8: Funkcija h v algoritmu Twofish. Vir: Prirejeno po: [17]

3.3.3.2 Besede podključev K_j

Besede, ki jih uporabimo pri vsaki rundi algoritma Twofish, izpeljemo s pomočjo funkcije h na sledeči način:

$$\begin{aligned}
&\rho = 2^{24} + 2^{16} + 2^8 + 1 \\
&A_i = h(2i\rho, M_e) \\
&B_i = h((2i+1)\rho, M_o) <<< 8 \\
&K_{2i} = (A_i + B_i) \mod 2^{32} \\
&K_{2i+1} = (A_i + 2B_i) \mod 2^{32} <<< 9.
\end{aligned}$$

3.3.3.3 Substitucijske tabele

Substitucijske tabele so definirane kot funkcija h brez množenja z matriko MDS. Celotno funkcijo g lahko torej definiramo kot:

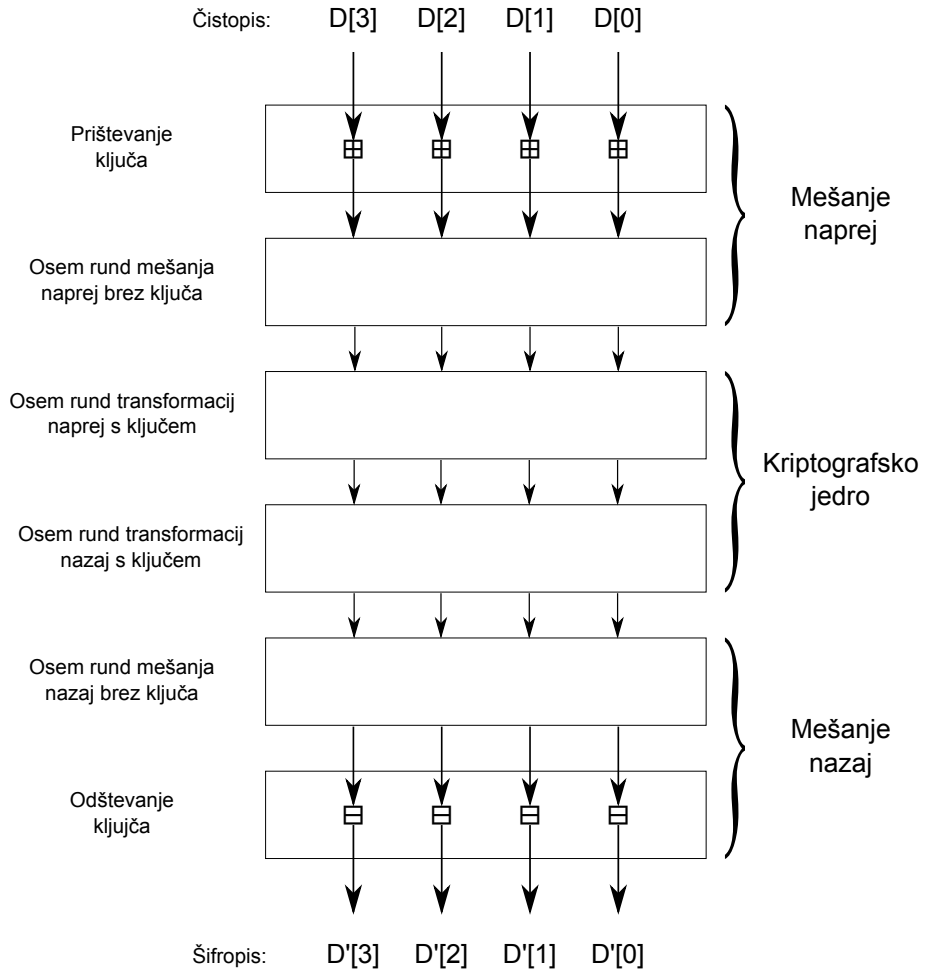
$$g(X) = h(X, S),$$

pri fiksnem S. Množenje z matriko MDS je namreč že definirano znotraj funkcije h.

3.4 MARS

MARS je algoritem, ki sestoji iz več plasti:

1. dodajanje ključa oziroma pred-beljenje,
2. 8 rund mešanja naprej brez ključa,
3. 8 rund transformacije naprej s ključem,
4. 8 rund transformacije nazaj s ključem,
5. 8 rund mešanja nazaj brez ključa
6. in odštevanje ključa ali post-beljenje. [20]



Slika 3.9: Algoritem MARS. Vir: Prirejeno po: [21]

Vseh 16 rund transformacij imenujemo kriptografsko jedro. V algoritmu je uporabljena 9×32 substitucijska tabela, ki jo v rundah mešanja obravnavamo kot dve 8×32 tabele.

V mešanjih in transformacijah uporabljamo tretji tip Feistelovega omrežja, ki deluje nad štirimi vhodi. Stanje je sestavljeno iz štirih besed. Shema algoritma je prikazana na sliki 3.9. Posamezne plasti algoritma so predstavljene v nadaljevanju.

3.4.1 Mešanje naprej

V tej fazi najprej seštejemo stanje in prve štiri besede podključev. Sledi osem rund modificiranega Feistelovega omrežja brez ključa.

V vsaki rundi uporabljamo eno izvirno besedo in tri ciljne besede. Uporabljamo dve 8×32 substitucijski tabeli $S0$ in $S1$, ki ju dobimo tako, da 9×32 substitucijsko tabelo S razbijemo na spodnji in zgornji del. Delovanje mešanja naprej prikazuje slika 3.10.

Znotraj ene runde razbijemo izvorno besedo na štiri bajte $b0$, $b1$, $b2$, $b3$. Najprej naredimo operacijo XOR med $S0[b0]$ in prvo ciljno besedo ter seštejemo $S1[b1]$ in prvo ciljno besedo. Nato seštejemo še $S0[b2]$ in drugo ciljno besedo ter naredimo operacijo XOR med $S1[b3]$ in tretjo ciljno besedo. Na koncu rotiramo izvorno besedo za 24 bitov v desno.

V naslednji rundi rotiramo besede tako, da prva ciljna beseda postane izvorna, druga postane prva, tretja postane druga in trenutna izvorna postane tretja ciljna.

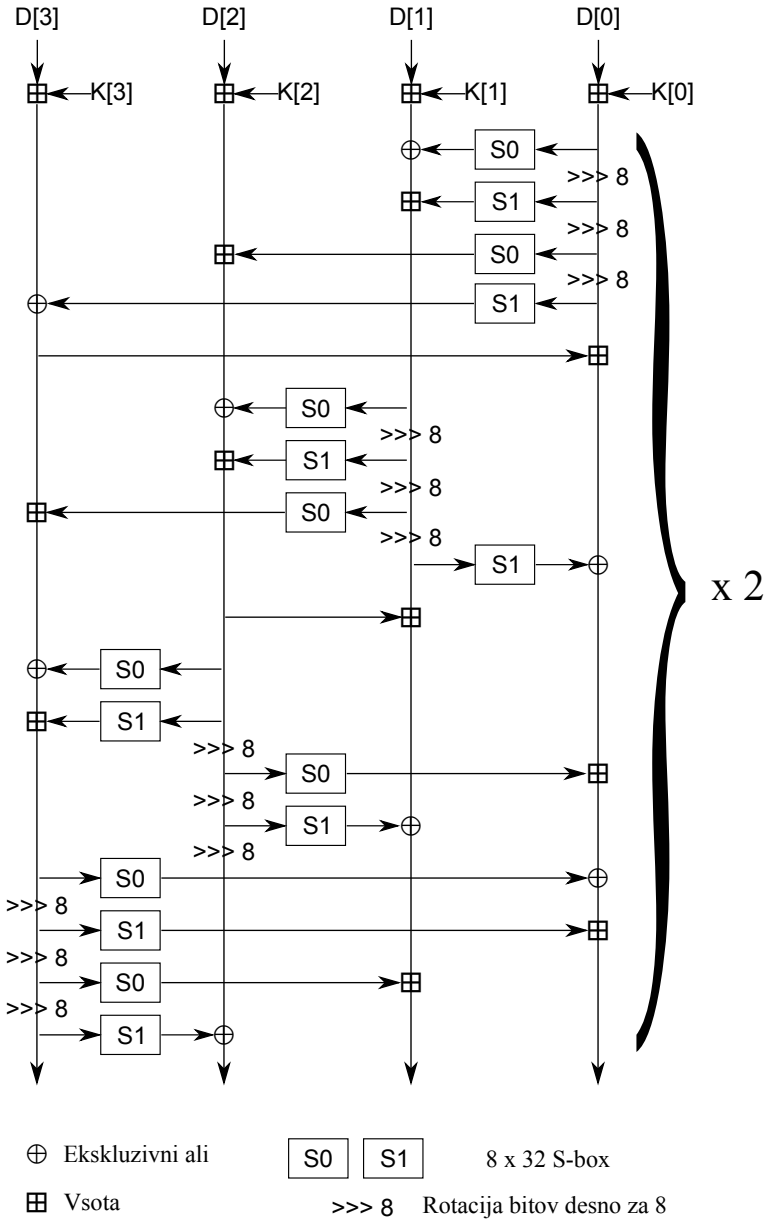
Poleg tega prištejemo v prvi in peti rundi tretjo ciljno besedo nazaj k izvorni ter v drugi in šesti rundi prvo ciljno.

Razlog za operacijo mešanje naprej je otežitev napada z izbranim čistopisom (ang. Chosen Ciphertext Attack). Poleg tega nam oteži zmanjševanje rund kriptografskega jedra pri linearnem in diferencialnem napadu [21].

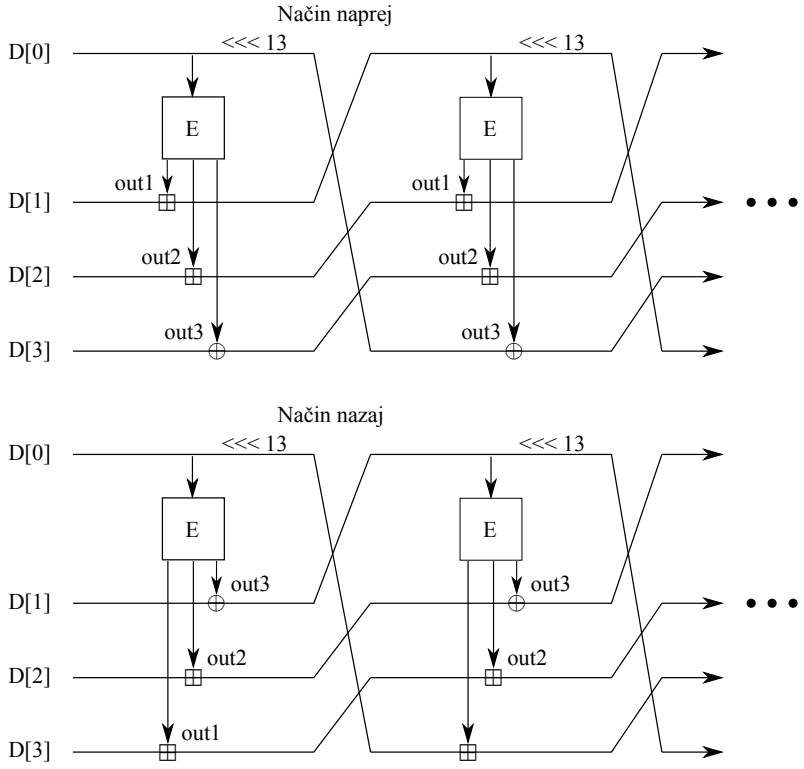
3.4.2 Kriptografsko jedro

Kriptografsko jedro je sestavljeno iz 16 rund Feistelovega omrežja. V vsaki rundi uporabimo ekspanzijsko funkcijo (funkcija E), ki sprejme besedo ter vrača tri besede. Zgradba Feistelovega omrežja je prikazana na sliki 3.11.

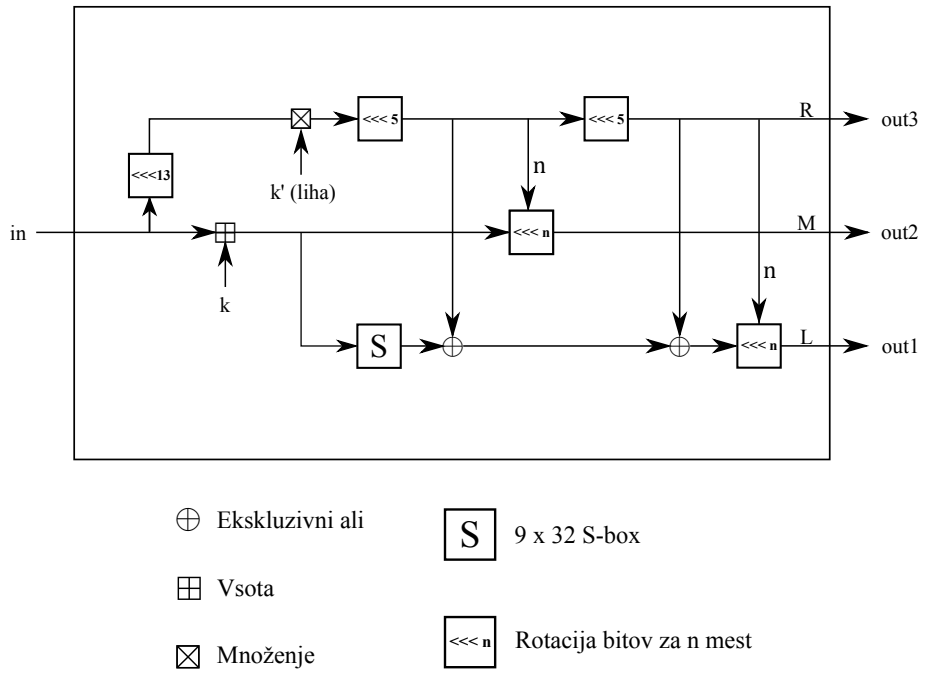
V vsaki iteraciji je izvorna beseda vhod v funkcijo E. Prvi izhod funkcije se XORa s prvo ciljno besedo, drugi z drugo ter tretji s tretjo. Na koncu se izvorna beseda rotira v levo za 13 bitov. Besede nato zamenjajo vrstni red kot pri mešanju naprej.



Slika 3.10: Mešanje naprej v algoritmu MARS. Vir: Prirejeno po [21]



Slika 3.11: Kriptografsko jedro v algoritmu MARS. Vir: Prirejeno po: [21]



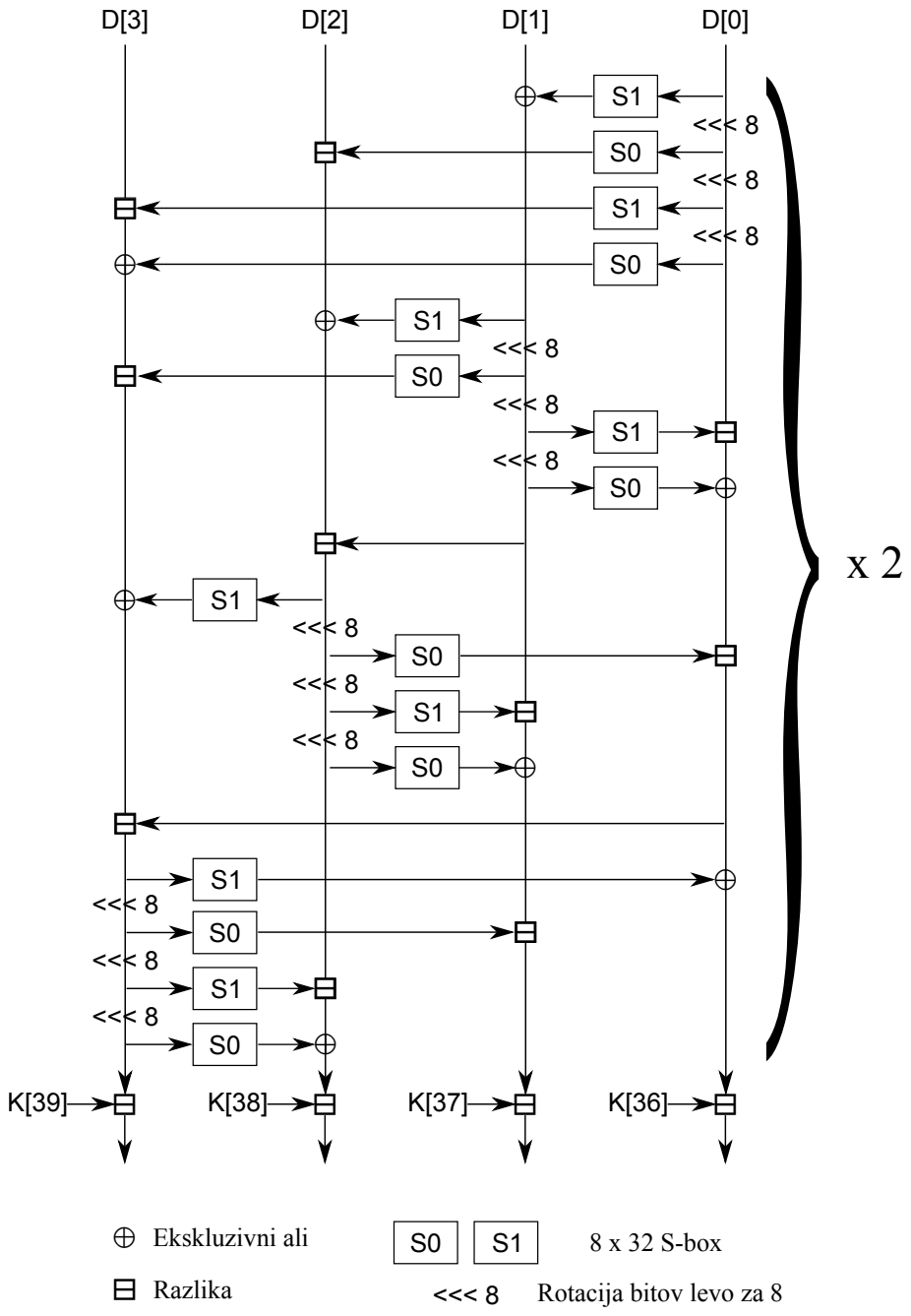
Slika 3.12: Funkcija `E` v kriptografskem jedru. Vir: Prirejeno po: [21]

3.4.2.1 Ekspanzijska funkcija

Ekspanzijska funkcija `E`, ki jo uporabljamo v kriptografskem jedru je shematsko prikazana na sliki 3.12. Funkcija na vhod sprejme eno besedo in vrača tri. Poleg vhodne besede potrebuje še dve besedi podključa, ki sta na sliki označeni kot `k` in `k'(liha)`. Funkcije zaradi zapletenosti ne bomo opisovali z besedo. Če želi bralec izvedeti več o samem načrtovanju funkcije, naj si prebere specifikacijo algoritma MARS. [21]

3.4.3 Mešanje nazaj

Mešanje nazaj je inverzna operacija mešanja naprej. Če bi dali izhod mešanja naprej na vhod mešanja nazaj, bi se operacije izničile. Shemo mešanja nazaj prikazuje slika 3.13.



Slika 3.13: Mešanje nazaj v algoritmu MARS. Vir: Prirejeno po: [21].

3.4.4 Razširjanje ključa

Razširjanje ključa nam mora iz seznama besed k, dolžine n ($n = 4$, 6 ali 8) zagotovi seznam besed razširjenega ključa K, dolžine 40. Ker se nekatere besede seznama K uporabljajo za množenje v funkciji E, nam mora procedura zagotoviti sledeče lastnosti:

- najnižja dva bita besed, sta nastavljena na 1,
- nobena izmed teh besed ne vsebuje 10 zaporednih 1 ali 0.

Procedura za razširjanje ključa poteka v naslednjih korakih:

1. Najprej kopiramo elemente seznama k v začasni seznam T, dolžine 15 besed. Nato zapišemo na mesto z indeksom n število n in dopolnimo z ničlami do zapolnjenosti:

$$\begin{aligned} T[0 \ldots n-1] &= k[0 \ldots n-1] \\ T[n] &= n \\ T[n+1 \ldots 14] &= 0 \end{aligned}$$

2. Nato v štirih iteracijah ponovimo:

   (a) Seznam T je transformiran s pomočjo linearne formule:

$$\begin{aligned} T[i] = T[i] \quad &\oplus \quad ((T[i-7 \mod 15] \\ &\oplus \quad T[i-2 \mod 15]) <<< 3) \\ &\oplus \quad (4i+j),\ i = 0 \ldots 14 \end{aligned}$$

   kjer j predstavlja iteracijo.

   (b) Premešamo seznam T:

$$T[i] = (T[i] + S[T[i-1 \mod 15] \mod 512]) <<< 9$$

   za $i = 0 \ldots 14$.

(c) Vzamemo 10 besed iz seznama T in jih preuredimo v naslednjih 10 besed razširjenega ključa K:

$$K[10j + i] = T[4i \mod 15],\ i = 0, \ldots, 9.$$

3. Nazadnje popravimo še tiste besede, ki jih uporabljamo pri množenju v funkciji E: $(K[5],\ K[7],\ \ldots,\ K[35])$, da imajo zahtevane lastnosti. Te besede popravimo na sledeči način:

   (a) Zapomnimo si spodnja dva bita $K[i]$: $j = K[i]\ \&\ 3$ in besedo, ki ima spodnja dva bita nastavljena na 1: $w = K[i] \mid 3$.

   (b) Sledi izdelava maske M, ki ima postavljene tiste bite, ki pripadajo zaporedni sekvenci 10 ali več 0 ali 1 v w:.

      i. Najprej postavimo bite maske M na 0. Nato postavimo M_j na 1, če in samo če w_j pripada sekvenci 10 ali več 0 ali 1 v w.

      ii. Postavimo na 0 tiste bite, ki predstavljajo konce sekvenc ali pa sta spodnja dva bita ali zgornji bit maske M.

   Avtorji so podali primer $w = 0^3 1^{13} 0^{12} 1011$, kjer x^i predstavlja i zaporednih bitov vrednosti x. Najprej izračunamo $M = 0^4 1^{25} 0^4$. Nato postavimo na 0 bite na položajih 4, 15, 16 in 28, da dobimo $M = 0^4 1^{11} 001^{10} 0^5$.

   (c) Uporabimo fiksno tabelo B dolžine štiri, da popravimo w. Elementi tabele se pravzaprav nahajajo znotraj tabele $S[265 \ldots 268]$. Zanje je značilno, da ne vsebujejo 10 zaporednih 0 ali 1. Vzamemo j iz koraka (a) za izbiro elementa v B ter spodnjih 5 bitov $K[i-1]$, da dobimo p:

$$p = B[j] <<< (K[i-1] \mod 32).$$

   (d) Na koncu izvedemo:

$$K[i] = w \oplus (p\ \&\ M),$$

da dobimo besedo razširjenega ključa. Ker sta spodnja dva bita v M postavljena na 0 in v w na 1 je zagotovljeno, da bosta spodnja dva bita $K[i]$ postavljena na 1. Zaradi lastnosti tabele B bo tudi zagotovljeno, da beseda $K[i]$ ne bo vsebovala 10 zaporednih 0 ali 1. [21]

3.5 RC6

RC6 je nadgradnja algoritma RC5 [18] in je v bistvu množica šifer. Množica pravimo zato, ker je vsaka šifra določena s tremi parametri: w, r in b. Določeno šifro znotraj množice imenujemo RC6-$w/r/b$. Parameter w pomeni dolžino besede, r pomeni število rund in b pomeni število bajtov ključa. V primeru izbora standarda AES so bili parametri $w = 32$, $r = 20$ in $b = 16$, 24 ali 32. Algoritem uporablja vrsto Feistelovega omrežja in je zaradi lažje analize načrtovan zelo preprosto.

Prilagojena verzija algoritma s parametri $w = 32$, $r = 20$ in $b = 16$, 24 ali 32 je prikazana s psevdokodo 3.6. Stanje sistema sestoji iz štirih besed: A, B, C in D. Pri tem velja opozoriti, da ukaz $(A, B, C, D) = (B, C, D, A)$ pomeni prirejanje ter da je S seznam besed podključev dolžine $2 \times r + 4$.

```
1  RC6Encrypt(A, B, C, D, S[2*20 + 4]){
2      B=B+S[0];
3      D=D+S[1];
4      for(i=1;i<=20;i++){
5          t=(B*(2*B+1))<<<5;
6          u=(D*(2*D+1))<<<5;
7          A=((A^t)<<<u)+S[i];
8          C=((C^u)<<<t)+S[2*i+1];
9          (A,B,C,D) = (B,C,D,A)
10     }
11     A=A+S[2*20+2];
12     C=C+S[2*20+3];
13 }
```

Psevdokoda 3.6: Psevdokoda algoritma RC6.

3.5.1 Razširjanje ključa

Razširjanje ključa je podobno kot pri $RC5$ in nam mora zagotoviti $2r + 4$ besed. Vhod v algoritem razvrščanja ključev je seznam besed L, dolžine c, kjer je njena vrednost izračunana po formuli $c = \lfloor b/4 \rfloor$. Seznam L dobimo tako, da seznam bajtov ključa pretvorimo v seznam besed po pravilu tankega konca. Izhod algoritma je seznam besed S, dolžine $2r+4$. Razširjanje ključa algoritma RC6 je prikazano v psevdokodi 3.7. V algoritmu smo uporabili dve konstanti $P32 = 0xB7E15163$ in $Q32 = 0x9E3779B9$. Prva je definirana kot decimalni del $e - 2$, druga pa kot decimalni del zlatega reza ϕ. [19]

```
1  KeySchedule(L[c], S[2*20 + 4]){
2      S[0] = P32;
3      for(i = 1; i < 2*20+4; i++)
4          S[i]=S[i-1]+Q32;
5      A=B=i=j=0;
6      v=3*max(c, 2*20+4);
7      for(s=1; s<v;s++){
8          A=S[i]=(S[i]+A+B) <<< 3;
9          B=L[j]=(L[j]+A+B) <<< (A+B);
10         i=(i+1) % (2*20+4);
11         j=(j+1) % c;
12     }
13 }
```

Psevdokoda 3.7: Razširjanje ključa algoritma RC6.

Poglavje 4

CUDA in OpenCL

To poglavje je namenjeno temu, da dobi bralec osnovno znanje o CUDI in OpenCL-ju, ki je potrebno za nadaljnje razumevanje implementacij vzporednih algoritmov. Najprej je predstavljena arhitektura CUDA in njen programski model. Na koncu sledi še predstavitev ogrodja za pisanje vzporednih programov za različne večnitne arhitekture OpenCL ter njegova uporaba z arhitekturo CUDA.

4.1 CUDA

CUDA (Compute Unified Device Architecture) je vzporedna računska platforma, ki omogoča uporabo grafičnih kartic za splošne namene (ang. General-Purpose GPU). To pomeni, da uporabimo grafični procesor za reševanje različnih vrst problemov in ne samo za izrisovanje grafike na računalnikov zaslon. Prvič je bila predstavljena leta 2007 in je namenjena izključno za Nvidijine grafične kartice. Za lažje razumevanje platforme bomo razdelili opis na strojni del, kjer je opisana arhitektura, hierarhijo pomnilnika, kjer so opisani različni tipi pomnilnikov in njihova uporaba ter programski model, ki opisuje kako programiramo CUDO.

4.1.1 Strojni model

GPE procesor z arhitekturo CUDA je sestavljen iz N multiprocesorjev (MP), ki vsebujejo po M tokovnih procesorjev (SP) ali jeder CUDA. Vsak multiprocesor poleg jeder vsebuje še množico 32-bitnih registrov, deljeni pomnilnik, ki je skupen enemu bloku niti in predpomnilnik.

CUDA uporablja t.i. SIMT (Single Instruction Multiple Thread) arhitekturo. Multiprocesor upravlja, razvršča in izvaja niti v skupinah po 32 niti, ki jim pravimo snop (ang. warp). Vse niti znotraj enega snopa začnejo na istem naslovu, a vsebujejo ločen programski števec in registre, kar omogoča vejitve in skoke. Kljub temu bo vzporednost niti zagotovljena samo kadar bodo vse niti znotraj enega snopa na isti poti. To pomeni, da vse niti sledijo istemu izvajanju ukazov. Ko pa imamo skoke in vejitve znotraj enega snopa, pa bo moral multiprocesor zaporedno izvesti različne poti.

4.1.2 Hierarhija pomnilnika

4.1.2.1 Globalni pomnilnik

Globalni pomnilnik se nahaja na RAM-u grafične kartice in predstavlja vmesnik med računalnikom in grafično kartico. Izmed naštetih pomnilnikov je zdaleč največji, a tudi najpočasnejši. Najpogosteje ga uporabimo zato, da prenesemo najprej podatke iz računalnikovega RAM-a v globalni pomnilnik ter nato iz njega v deljeni pomnilnik ali registre, kjer se lahko podatki obdelajo hitreje. V določenih primerih ga uporabimo tudi za komunikacijo med različnimi bloki niti.

Dostop do globalnega pomnilnika se vrši s transakcijami po 32, 64, ali 128 biti naenkrat. Zaradi tega morajo biti podatkovne strukture v globalnem pomnilniku poravnane glede na velikost transakcije. Poleg tega je dostop do globalnega pomnilnika najbolj učinkovit, kadar je zaporeden (ang. coalesced access). To pomeni, da bo vsaka zaporedna nit dostopala do podatka z zaporednim naslovom, zato bomo lahko z eno transakcijo pridobili podatke za več niti hkrati.

4.1.2.2 Deljeni pomnilnik

Deljeni pomnilnik se nahaja na vsakem multiprocesorju in je skupen enemu bloku niti. Uporabljamo ga za komunikacij med niti enega bloka in za hitrejšo obdelavo podatkov. Je hitrejši od globalnega pomnilnika in služi kot nekakšen predpomnilnik, ki ga upravljamo sami.

Njegova hitrost je dosežena s tem, da je fizično na multiprocesorji in da je razdeljen na enote, ki jih imenujemo banke, do katerih lahko niti dostopajo vzporedno. Deljeni pomnilnik je za en snop razdeljen na 32 bank, kjer je vsaka banka velika štiri bajte ali eno besedo. Če shranimo v deljeni pomnilnik več kot 32 besed podatkov, so ti porazdeljeni v banke po modulu 32.

Vsaka banka lahko obdela eno zahtevo na cikel neodvisno od ostalih bank. Kadar vsaka izmed 32 niti znotraj enega snopa dostopa do svoje banke, imamo vzporedni dostop do vseh bank naenkrat.

Podobeno učinkovit način dostopa imamo, ko vse niti znotraj snopa dostopajo do iste banke (broadcast), saj se v tem primeru v resnici zgodi samo en dostop.

Težava nastane, kadar imamo vzorec dostopa, ki je nekaj vmes med zgornjima dvema. V tem primeru pride do konflikta. Kadar recimo dve niti v paru znotraj istega snopa dostopajo do iste banke, imamo dvosmerni konflikt (ang. two-way conflict). V tem primeru se bo dostop naredil v dveh ciklih, saj bo ena nit morala čakati na drugo. Več ko imamo niti, ki dostopajo do iste banke, slabše je vse dokler vseh 32 niti spet ne dostopa do iste banke. [22]

4.1.2.3 Registri

Registri se nahajajo na vsakem multiprocesorju in so najhitrejša oblika pomnilnika na arhitekturi CUDA. Registri so lokalni na ravni ene niti in jih uporabljamo za podatke, nad katerimi izvajamo veliko računanja. V njih se običajno shranijo lokalne spremenljivke.

Največja težava registrov je, da jih imamo omejeno število na eno nit in na en multiprocesor. Kadar eni niti zmanjka registrov, mora podatke shraniti

na veliko počasnejši lokalni pomnilnik.

4.1.2.4 Lokalni pomnilnik

Lokalni pomnilnik je del globalnega pomnilnika, ki je lokalen na ravni ene niti. Zaradi njegove počasnosti se mu poskušamo v največji meri izogniti. Prevajalnik bo lokalni pomnilnik ponavadi izkoristil takrat, ko imamo preveč uporabljenih registrov na eno nit ali pa uporabljamo prevelike lokalne strukture in sezname, ki jih ni mogoče shraniti zgolj v registrih.

4.1.2.5 Konstantni pomnilnik

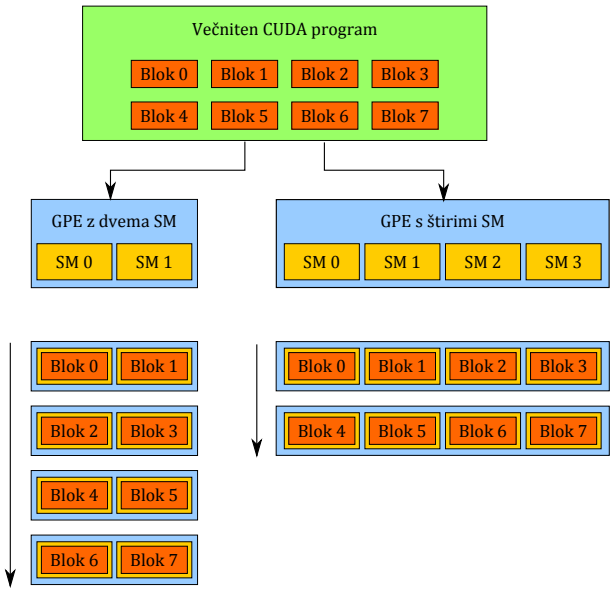
Konstantni pomnilnik je poseben prostor na globalnem pomnilniku, ki se predpomni na vsakem multiprocesorju. Zaradi tega je hitrejši, a samo če vse ali večino niti dostopa do istega podatka. Poleg tega ima to omejitev, da je samo pisalni pomnilnik.

4.1.3 Programski model

Programe za CUDO pišemo v programskem jeziku CUDA C, ki je razširitev programskega jezika C. Za prevajanje programov napisanih v CUDA C uporabljamo Nvidia Cuda Compiler (NVCC), ki poskrbi za ločeno prevajanje zaporedne kode napisane v C-ju in vzporednih funkcij, ki jim pravimo ščepci (ang. kernel). Ščepci se prevedejo najprej v vmesno kodo PTX, ki jo gonilnik naprave prevede v ustrezno strojno kodo v času zagona aplikacije.

Ščepci so posebne vrste funkcij, ki se od ostale programske kode razlikujejo v tem, da se izvedejo na grafični kartici. Predstavljajo torej naloge, ki naj jih opravi grafična kartica. Izvedel se bo tolikokrat vzporedno, kolikor imamo definiranih niti za neko opravilo.

Ščepci imajo v programski kodi deklaracijo `__global__`, kadar jih želimo klicati iz gostiteljeve kode. Ko pa imajo deklaracijo `__device__`, pa jih lahko kličemo le iz nekega drugega ščepca. Ščepec poženemo iz gostiteljeve kode tako, da pokličemo `kernelName <<<dimGrid, dimBlock`



Slika 4.1: Avtomatska razporeditev blokov niti med multiprocesorji. Vir: Prirejeno po: [22].

`>>>(arg0, arg1, ...)`, kjer `dimGrid` in `dimBlock` predstavljata velikost mreže in bloka niti.

Posamezne niti so združene v bloke niti in ti naprej v mrežo (ang. grid). Vsaka nit se znotraj enega bloka identificira s tremi koordinatami. Prav tako se vsak blok identificira s tremi koordinatami znotraj mreže. Vsaka nit se torej globalno identificira s svojo koordinato znotraj bloka in koordinato bloka.

Vsaka izmed niti se na strojnem nivoju izvede na enem jedru CUDA, medtem ko se na enem multiprocesorju naenkrat izvede en blok. Število hkrati izvedenih blokov pa je odvisno od števila multiprocesorjev, kot to prikazuje slika 4.1.

4.2 OpenCL

OpenCL (Open Computing Language) je ogrodje za pisanje programov za vzporedne platforme. Za razliko od CUDA C, ki je namenjena samo za

CUDA	**OpenCL**
Nit (ang. Thread)	Delovni predmet (ang. Work-item)
Blok niti (ang. Thread block)	Delovna skupina (ang. Work-group)
Globalni pomnilnik	Globalni pomnilnik
Konstantni pomnilnik	Konstantni pomnilnik
Deljeni pomnilnik	Lokalni pomnilnik
Lokalni pomnilnik	Zasebni pomnilnik

Tabela 4.1: Preslikava terminologije med CUDO in OpenCL.

grafične kartice z arhitekturo CUDA, je programe v OpenCL-ju možno pognati na različnih arhitekturah. Sem spadajo več jedrni procesorji, grafične kartice različnih proizvajalcev, procesorjev za digitalno procesiranje signalov, itd.

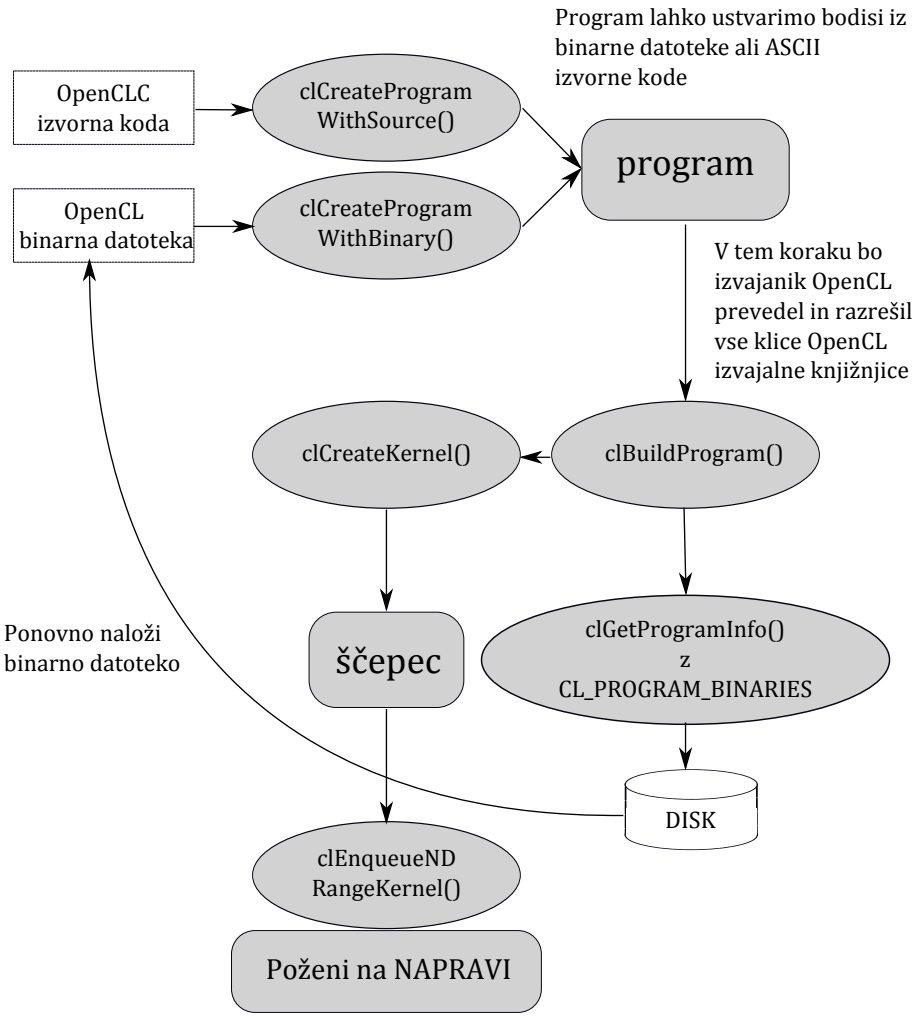
Pri snovanju OpenCL-ja so se močno zgledovali po arhitekturi CUDA in njenih rešitvah. Zaradi tega je programe, ki jih že imamo napisane v CUDA C, enostavno spremeniti v programe napisane v OpenCL. To velja še posebej takrat kadar uporabljamo strojno opremo z arhitekturo CUDA. Obstaja namreč preslikava večine pojmov in sistemskih klicev med CUDO C in OpenCL. Prav tako je uporabljena enaka hierarhija pomnilnikov kot pri CUDI [32]. Nekaj primerov preslikave med terminologijama CUDA in OpenCL je prikazanih v tabeli 4.1.

Slaba stran OpenCL-ja je, da je veliko več režije kot na platformi CUDA. Razlog za to je prenosljivost programa napisanega v OpenCL C med različnimi arhitekturami. Zaradi tega moramo znotraj programa vedno najprej izbrati platformo kot je recimo NVIDIA, AMD ali Intel. Nato sledi izbira naprave, saj imamo lahko več vrst naprav v računalniku iste platforme. Na koncu ustvarimo še kontekst, ki predstavlja nekakšno okolje, ki vključuje napravo in ostale objekte kot so vrste, programi itd.

Poleg tega moramo znotraj programske kode sami poskrbeti za prevajanje programa, ki se bo izvedel na neki napravi. Program moramo zato imeti shranjen v nekem nizu ali zunanji datoteki. Obstajata dva načina za prevajanje programa. Prvi je, da programsko kodo preberemo iz datoteke ali

niza ter jo prevedemo. Drugi način pa, da preberemo programsko kodo samo enkrat, jo prevedemo in shranimo kot binarno datoteko. Kadar prevajamo za arhitekturo CUDO je izhod vmesna koda PTX, ki je ustvarjena s prevajalnikom NVCC. Slednji način smo izbrali tudi v naših implementacijah, saj se z njim izognemo dodatnemu koraku prevajanja.

Sledi še ustvarjanje objekta, ki mu v OpenCL terminologiji pravimo ščepec in je nekakšno JIT prevajanje na CUDI. Celoten postopek prevajanja programa in nato še prevajanje v ščepec prikazuje slika 4.2. [23]



Slika 4.2: Ustvarjanje objektov program in ščepec z OpenCL Vir: Prirejeno po: [23].

Poglavje 5

Zaporedne implementacije

To poglavje je namenjeno optimalnim zaporednim implementacijam algoritmov iz poglavja 3. Opisani so vzorci in izboljšave, ki smo jih uporabili za doseganje čim višje prepustnosti.

Zgledovali smo se po že obstoječih implementacijah za 32-bitne procesorje. Pregledali smo optimalne implementacije avtorjev algoritma, ki so jih morali poslati skupaj s specifikacijo na izbor AES. Prav tako smo pregledali nekaj že obstoječih implementacij kot je recimo knjižnica Crypto++ za programski jezik C++, GNU Crypto v Javi in CryptoPlus v Pythonu. Pri vseh knjižnicah smo opazili podobne vzorce implementacije.

Za šifriranje večih blokov smo uporabili način CTR, ki smo ga opisali pri definiciji bločnih šifer 2. Pri tem načinu potrebujemo za šifriranje čistopisa in dešifriranje šifropisa samo šifrirno funkcijo, ki ustvari tok ključev oziroma šifrira števce.

5.1 Rijndael

Občutno izboljšavo algoritma Rijndael podajo že sami avtorji algoritma. Predlagajo združitev operacij `SubBytes`, `ShiftRow` in `MixColumns` v štiri substitucijske tabele. Namesto ene 8×8 tabele, imamo sedaj štiri 8×32 tabele in eno 8×32 tabelo za zadnjo rundo in razširjanje ključa.

Postopek za izračun štirih tabel je sledeč. Stolpec j neke runde, je po operaciji `AddRoundKey` enak:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix},$$

kjer je vektor d_j definiran kot izhod operacije $MixColumns$

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}.$$

Vektor c_j je definiran kot izhod operacije $ShifrRow$:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-1} \\ b_{2,j-2} \\ b_{3,j-3} \end{bmatrix}$$

in

$$b_{i,j} = S[a_{i,j}].$$

Če vse skupaj zložimo nazaj, dobimo enačbo

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j-1}] \\ S[a_{2,j-2}] \\ S[a_{3,j-3}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}.$$

Množenje lahko nato zapišemo kot linearno kombinacijo vektorjev:

$$S[a_{0,j}] \begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix} \oplus S[a_{0,j-1}] \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix} \oplus S[a_{0,j-2}] \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix} \oplus S[a_{0,j-3}] \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix}.$$

Kar pa lahko izračunamo vnaprej v štiri nove substitucijske tabele:

$$T_0[a] = \begin{bmatrix} S[a] \cdot 2 \\ S[a] \\ S[a] \\ S[a] \cdot 3 \end{bmatrix} T_1[a] = \begin{bmatrix} S[a] \cdot 3 \\ S[a] \cdot 2 \\ S[a] \\ S[a] \end{bmatrix} T_2[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 3 \\ S[a] \cdot 2 \\ S[a] \end{bmatrix} T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \cdot 3 \\ S[a] \cdot 2 \end{bmatrix}.$$

Računanje enega stolpca stanja se zato poenostavi v štiri poizvedbe in štiri operacije XOR.

$$e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{1,j-2}] \oplus T_3[a_{1,j-3}] \oplus k_j.$$

Za shranjevanje vseh štirih tabel potrebujemo 4 KB prostora. Avtorji sicer pravijo, da se je temu možno izogniti z rotacijo ene tabele, kadar uporabljamo arhitekturo s premalo pomnilnika.

V zadnji rundi ne računamo $MixColumns$. Zaradi tega potrebujemo še zadnjo tabelo, ki ima izračunane samo vrednosti navadne tabele S.

5.2 Serpent

Učinkovita implementacija algoritma Serpent je odvisna predvsem od števila bitnih operacij, zamikov in prirejanj, ki jih moramo izvesti za izračun poizvedbe neke tabele. Eden izmed načinov, za optimizacijo izračuna je preiskovanje prostora kombinacij operacij, ki nam dajo želen rezultat. To pomeni, da poiščemo najkrajšo možno zaporedje operacij, ki nam da enak rezultat kot originalna tabela.

Takšnega pristopa sta se lotila Simpson in Gladman [37], ki sta s pomočjo gruče računalnikov dosegla pohitritev izvedbe tabel tabel. Podobno je naredil tudi Osvik [24], le da je postavil bolj striktne pogoje pri iskanju zaporedja. Eden izmed pogojev je bil tudi omejitev uporabe petih registrov, saj je opazil, da Gladmanove tabele uporabljajo veliko začasnih spremenljivk. To na arhitekturah z omejenim številom registrov kot sta recimo x86 ali CUDA, povzroči prenašanje podatkov v pomnilnik in nazaj. S tem pristopom mu je uspelo algoritem močno pohitriti v primerjavi z Gladmanom.

5.3 Twofish

Pri algoritmu Twofish je funkcija g sestavljena iz štirih tabel in množenja z matriko MDS. Ker so tabele odvisne od vhodnega ključa, je posledično tudi funkcija g odvisna od vhodnega ključa.

Isti ključ lahko uporabimo za šifriranje več blokov, zato se zdi smiselno, da del funkcije g izračunamo vnaprej v algoritmu za razširjanje ključa. Na ta način se tudi pohitri šifrirni algoritem.

Avtorji algoritma Twofish zato predlagajo različne načine izračunanja funkcije g vnaprej:

- **Full Keying:** Pri tem načinu izračunamo štiri substitucijske tabele in jih združimo z množenjem z matriko MDS. S tem dobimo štiri 8×32 tabele, ki zasedejo skupaj 4Kb prostora. Izračun funkcije g se poenostavi na štiri poizvedbe in tri operacij XOR. Ta način je priporočljiv takrat, kadar se isti ključ uporabi za šifriranje velikega števila blokov.

- **Partial Keying:** Pri tem načinu izračunamo štiri 8×8 substitucijske tabele, a potrebujemo zato še štiri fiksne 8×32 tabele za množenje z matriko MDS. Pri tem načinu zmanjšamo velikost tabel na 1Kb. Izračun funkcije g je sedaj sestavljeno iz štirih poizvedb v tabele in štirih poizvedb v tabele MDS.

- **Minimal Keying:** Ta način je izveden tako, da se izračuna ena plast

permutacij q manj v funkciji h. Preostanek se izračuna pri samem šifriranju.

- **Zero Keying:** Pri tem načinu izračunamo funkcijo $g(X) = h(X, S)$ znotraj šifriranja brez poizvedb v tabele. Zato je ta način tudi najpočasnejši in primeren samo, kadar nimamo zadosti pomnilnika.

Ker bomo šifrirali veliko blokov, smo izbrali način Full Keying.

Poleg te izboljšave smo uporabili tudi izboljšano množenje z matriko RS, ki za množenje uporablja generatorski polinom matrike.

5.4 MARS

Algoritem MARS smo poenostavili tako, da smo v kriptografskem jedru uporabili samo eno makro funkcijo za Forward in Backwards način. To naredimo tako, da ustrezno zamenjamo vrstni red argumentov kot pri algoritmu RC6. Poleg tega smo eno rundo Feistelovega omrežja in funkcijo E združili, saj smo opazili, da se velik del kode prepleta.

5.5 RC6

Pri algoritmu RC6 nismo zasledili kakšnih posebnih izboljšav algoritma, saj je dovolj preprost za učinkovito implementacijo na večini procesorjev. Omenimo lahko edino, da smo uporabili preprost način za zamenjavo vrednosti $(A, B, C, D) = (B, C, D, A)$. To naredimo tako, da smo definirali funkcijo runde kot makro funkcijo v C-ju ter nato samo zamenjali vrstni red vhodnih argumentov v vsaki rundi. Preprocesor nato poskrbi, da se makro funkcija prevede v običajno programsko kodo z ustreznim vrstnim redom.

Poglavje 6

Vzporedne implementacije

V tem poglavju so predstavljene optimalne vzporedne implementacije finalistov AES. Najprej je predstavljen navaden vzporedni algoritem na osnovi razporejanja podatkov, ki ga uporabimo za vzporedno implementacijo vseh petih finalistov. Deluje tako, da podatke enakomerno porazdeli med niti. Nato sledi še predstavitev vzporednih implementacij z bitnimi rezinami finalistov Rijndael in Serpent.

6.1 Vzporedne implementacije na osnovi razporejanja podatkov

Ker je delovanje vseh petih implementacij na osnovi razporejanja podatkov podobnih, bomo v tem delu poglavja opisali samo vzporedni vzorec oziroma kako naredimo bločno šifro vzporedno.

Bločne šifre delujejo tako, da kot vhod vzamejo vhodni ključ in blok podatkov ter ga šifrirajo. Če imamo za več kot en blok podatkov, uporabimo različne načine delovanja, ki smo jih opisali v poglavju 2.

Izmed naštetih načinov sta ECB in CTR edina, ki omogočata neodvisno vzporedno šifriranje blokov podatkov. Zaradi tega je prva ideja, ki nam pride na misel, da uporabimo enega izmed teh dveh načinov, en blok podatkov pa obdela ena ali več niti. V našem primeru bomo uporabili način CTR, saj

je varnejši od načina ECB, ki je ranljiv na napad z izbranim čistopisom (ang. Choosen Plaintext Attacks). Poleg tega je ta način enostavnejši za implementacijo, saj potrebujemo zgolj šifrirno funkcijo algoritma za šifriranje in dešifriranje.

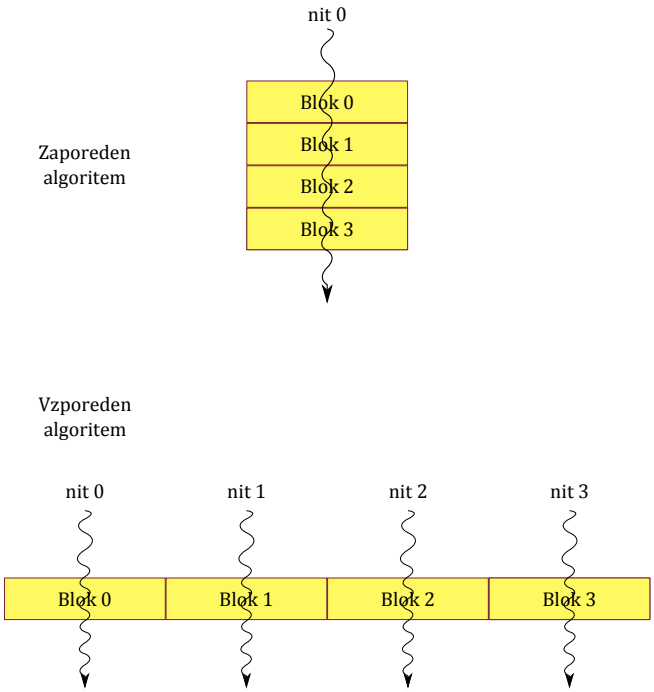
Prvo vprašanje, ki se nam postavi je, koliko niti naj obdeluje en blok podatkov oziroma kako podatke razdeliti med niti. Raziskavo na to temo so naredili Iwai in dr. [6], ki v svojem delu primerjajo različne načine vzporedne implementacije algoritma Rijndael. Narejena je bila primerjava, kakšna porazdelitev podatkov na nit je najbolj optimalna oziroma koliko niti naj obdeluje en blok podatkov. Prišli so do zaključka, da je najučinkovitejša izvedba takrat, kadar ena nit obdeluje po en blok podatkov naenkrat. Razlogi za to so neodvisnost ene niti od druge in da ni potrebe po sinhronizaciji ter njihovemu razhajanju (ang. thread divergency). Podoben pristop so uporabili tudi Li in dr. [8] in pokazali primerjavo z nekaterimi ostalimi implementacijami ter dosegli solidno prepustnost. Razdelitev podatkov med niti prikazuje slika 6.1

Način CTR deluje tako, da najprej ustvarimo števce, jih šifriramo z izbrano bločno šifro, da dobimo tok ključev ter na koncu naredimo še XOR med tokom ključev in čistopisom, s čimer dobimo šifropis.

Za ustvarjanje števcev smo uporabili enega izmed načinov, ki jih priporoča NIST [25]. Števci so ustvarjeni tako, da nam prvih 64 bitov predstavlja naključni žeton (ang. nounce), ki je skupen vsem šifriranim blokom. Ostalih 64 blokov predstavlja zaporedno številko šifriranega bloka, ki jo dobimo na podlagi globalnega indeksa niti, saj vsaka nit skrbi za svoj blok podatkov.

Poleg glavnega šifrirnega algoritma, je del bločne šifre tudi algoritem za razširjanje ključa. Med vsemi petimi finalisti opazimo, da je ta algoritem iterativen, saj za izračun naslednjega podključa potrebujemo izračunan najmanj prejšnji podključ. Zaradi tega sta nam na voljo dve rešitvi. Ali razširjanje ključev izvedemo na CPE in jih nato prenesemo na grafično kartico ali pa vsaka nit izračuna podključe posebej.

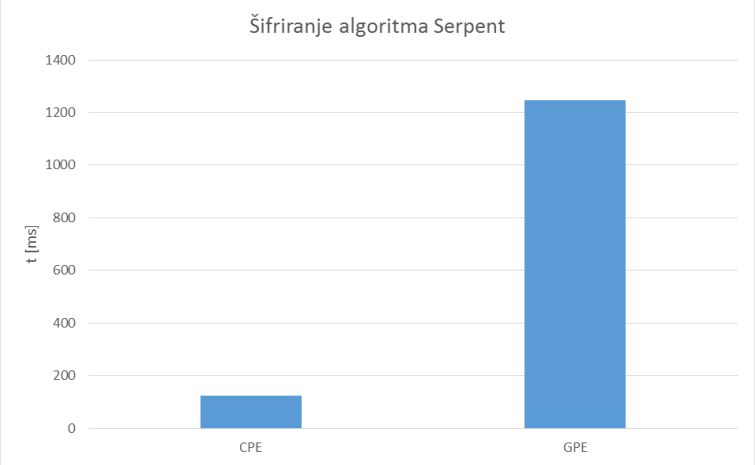
Ker bi šlo v drugem primeru za podvojeno računanje istih podatkov na



Slika 6.1: Razdelitev podatkov med več niti.

različnih nitih, smo se odločili za prvo rešitev. Druga rešitev bi se izkazala za učinkovito le, če bi za vsak blok uporabili različen vhodni ključ. Dodatna slabost, ki jo prinese druga rešitev je, da bi niti v tem primeru hranile podključe v lokalnem pomnilniku namesto konstantnem, kar bi še dodatno upočasnilo algoritem.

Našo izbiro smo potrdili s preizkusom, ki ga prikazuje slika 6.2. Na njej je prikazan povprečni čas šifriranja za algoritem Serpent, kadar razširjamo ključe na CPE in kadar to naredi vsaka nit posebej. Preizkusili smo z naključnimi podatki velikosti 128 MiB in naključnim 128 bitnim ključem. Iz slike je razvidno, da je prva rešitev veliko hitrejša. Podoben rezultat je pričakovan tudi pri ostalih algoritmih, saj ima Serpent relativno enostavno razširjanje ključa.



Slika 6.2: Primerjava šifriranja algoritma Serpent na CUDI, kadar je razširjanje ključev narejeno na CPE in na GPE.

6.1.1 Časovna zahtevnost in pohitritev

Sedaj, ko imamo definiran vzporedni algoritem, se lahko vprašamo kakšna je njegova časovna zahtevnost in kakšno pohitritev smo dosegli v primerjavi z zaporednim algoritmom.

Da lahko to izračunamo, moramo najprej definirati od katerega vhodnega parametra je odvisna časovna zahtevnost. Najbolj smiseln vhodni parameter se nam zdi število blokov, saj je to najmanjša enota, ki jo lahko obdela bločna šifra naenkrat. Ta parameter bomo označili z n.

Časovna zahtevnost zaporednega algoritma je enaka vsoti časovnih zahtevnosti algoritma za razširjanje ključa in šifriranja blokov. Algoritem za razširjanje ključa ima vedno konstantno časovno zahtevnost C_1, ne glede na to, koliko blokov bomo šifrirali. Čas zaporedne bločne šifre je sestavljen iz šifriranja n blokov. Ker uporabljamo način CTR, je šifriranje enega bloka sestavljeno iz ustvarjanja števcev, šifriranje števcev in operacije XOR med tokom in čistopisom. Ustvarjanje števca za en blok je konstantna operacija C_2. Šifriranje števca je prav tako konstantna operacija C_3, saj je velikost enega bloka vedno enaka ne glede na število blokov. C_4 pa je konstantna operacija XOR med tokom in čistopisom. Časovna zahtevnost zaporedne

bločne šifre je zato:

$$T_1(n) = C_1 + n(C_2 + C_3 + C_4) = C_1 + nC_5 = \theta(n)$$

Za izračun pohitritve potrebujemo še časovno zahtevnost vzporednega algoritma. To bomo izračunali za procesor s P jedri, ki je zmožen tvoriti P niti in teoretični procesor z neskončno jedri. Algoritma za razširjanje ključa nismo naredili vzporednega, a to ni težava, saj je njegova časovna zahtevnost konstantna. Šifrirni algoritem, ki v zaporednem primeru šifrira n-krat zaporedoma, pa smo razdelili med P delavci oziroma niti. Ker se časovna zahtevnost vzporednega šifrirnega algoritma načeloma razlikuje od zaporednega, moramo uporabiti novo konstanto C_6. Časovna zahtevnost za procesor s P jedri je zato:

$$T_P(n) = C_1 + \frac{n}{P} \cdot C_6 = \theta(n)$$

Iz računa vidimo, da je časovna zahtevnost še vedno linearna, a z manjšo konstanto.

V primeru, da imamo teoretični procesor z neskončno jedri, lahko predpostavimo, da imamo neskončno niti, kjer vsaka nit obdela svoj blok podatkov. V tem primeru bo čas šifriranja enak kot če bi šifrirali samo en blok podatkov. Časovna zahtevnost je v tem primeru enaka:

$$T_\infty(n) = C_1 + C_6 = \theta(1)$$

Sedaj, ko imamo vse časovne zahtevnost, lahko izračunamo pohitritev in vzporednost. Pohitritev je definirana kot količnik časovne zahtevnosti zaporednega algoritma in vzporednega algoritma na P procesorjih. V našem primeru je ta vrednost enaka:

$$\frac{T_1(n)}{T_P(n)} = \frac{C_1 + n \cdot C_5}{C_1 + \frac{n}{P} \cdot C_6}$$

Ker je čas razširjanja ključa veliko manjši kot šifriranje n blokov, ga lahko izpustimo:

$$\frac{T_1(n)}{T_P(n)} = \frac{C_1 + n \cdot C_5}{C_1 + \frac{n}{P} \cdot C_6} \approx \frac{n \cdot C_5}{\frac{n}{P} \cdot C_6} \approx P \cdot C_7$$

Iz računa vidimo, da ima naš vzporedni algoritem perfektno linearno pohitritev.

Vzporednost je količnik med zaporedno časovno zahtevnostjo in časovno zahtevnostjo vzporednega algoritma na procesorju z neskončno jedri. Ta vrednost nam pove, kolikšna je maksimalna možna pohitritev nekega algoritma, oziroma koliko niti moramo uporabiti, da bo delo konstantno. Količnik je v našem primeru enak:

$$\frac{T_1(n)}{T_\infty(n)} = \frac{C_1 + n \cdot C_6}{C_1 + C_5} \approx n \cdot C_7 \approx \theta(n)$$

Ta vrednost nam pove, da potrebujemo n niti za maksimalno pohitritev algoritma pri vhodu velikem n blokov.

Naš izračun je kar se da splošen in velja tako za grafične kartice kot za navadne procesorje z več jedri.

6.1.2 Prenos in hranjenje podatkov

V tem delu poglavja je razloženo in utemeljeno kje se nahaja kateri izmed podatkov, ki jih uporablja naš šifrirni algoritem in kako podatke prenesemo v ta pomnilnik.

6.1.2.1 Čistopis in šifropis

Če hočemo šifrirati čistopis na grafični kartici, ga moramo najprej prenesti iz pomnilnika računalnika v pomnilnik grafične kartice ter na koncu nazaj kot šifropis. Za ta namen je uporabljen globalni pomnilnik, ki predstavlja vmesnik med CPE-jem in grafično kartico. Nahaja se izven procesorja grafične kartice, zaradi česar je počasen.

Ponavadi poteka postopek prenosa podatkov v in iz globalnega pomnilnika tako, da najprej rezerviramo prostor, nato prenesemo podatke, sledi obdelava podatkov ter na koncu še prenos iz globalnega pomnilnika in sprostitev prostora.

Kadar pa dostopamo do globalnega pomnilnika samo na začetku ali na koncu jedra, je smiselno uporabiti zero-copy ali preslikan pomnilnik. Ker so naši algoritmi takšne vrste, ga uporabljamo tudi mi.

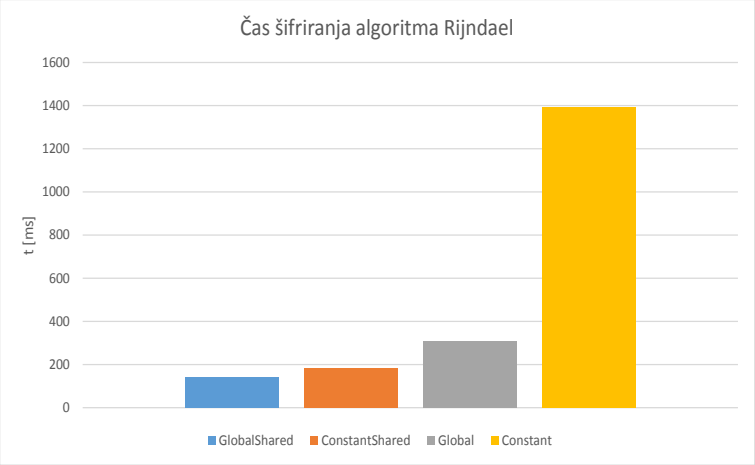
Ob uporabi preslikanega pomnilnika se podatki, ki jih imamo v pomnilniku računalnika, neposredno uporabijo na grafični kartici. Poleg tega nam omogoča, da se manjši kosi podatkov hkrati prenašajo in obdelujejo vzporedno kot pri cevovodu [22].

Pogoj, ki ga zahteva preslikan pomnilnik, je zaporedni dostop niti do globalnega pomnilnika. Poenostavljeno povedano to pomeni, da vsaka nit dostopa do svojega zaporednega naslova. V naših algoritmih mora vsaka nit prebrati 128 bitov podatkov ali en blok podatkov. Če beremo teh 128 bitov kot štiri zaporedna branja štirih besed, nastanejo vrzeli pri naslavljanju. V tem primeru nimamo zaporednega dostopa do podatkov in bi potrebovali štiri transakcije na vsako nit. Boljši način je uporaba vektorskega tipa `uint4`, ki povzroči hkratno branje štirih besed. Posledično imamo v tem primeru zaporedni dostop, kar je možno realizirati z eno samo transakcijo na nit. Ko je čistopis enkrat prebran iz pomnilnika, ga shranimo v štiri 32-bitne registre.

6.1.2.2 Substitucijske tabele

Pet finalistov lahko razdelimo v dve kategoriji. Takšne, ki za delovanje potrebujejo substitucijske tabele v spominu in takšne, ki tega ne potrebujejo. V prvo kategorijo spadajo Rijndael, Twofish in MARS, v drugo po Serpent in RC6. Pri implementaciji algoritmov iz prve kategorije smo morali upoštevati, da je potrebno tabele nekako prenesti iz pomnilnika računalnika v pomnilnik grafične kartice.

Za hranjenje tabel, smo izbrali deljeni pomnilnik, saj se nahaja na vezju grafične kartice in je zaradi tega najhitrejši. To odločitev je podkrepljena v

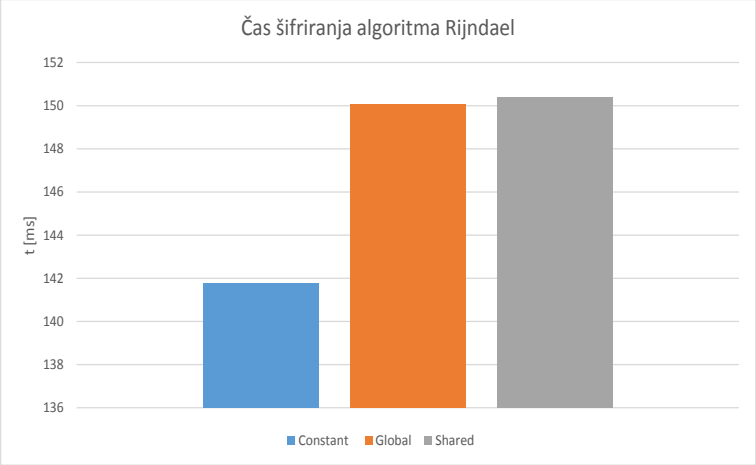


Slika 6.3: Čaš šifriranja algoritma Rijndael pri različnih lokacijah substitucijskih tabel.

številnih člankih kot so recimo [4, 7, 26, 9] in [8]. Poleg tega pa smo za potrditev te odločitve naredili preizkus, kjer tabele prenašamo in hranimo v različnih vrstah pomnilnika.

Preizkusili smo štiri različne načina prenosa in uporabe substitucijskih tabel. Pri prvem načinu se tabele nahajajo v globalnem pomnilniku, pri drugem načinu jih imamo v konstantnem, pri tretjem načinu tabele najprej prenesemo iz računalnika na grafično kartico v globalnem pomnilniku, nato pa jih prestavimo v hitrejši deljenem pomnilnik, pri četrtem preizkusu pa tabele najprej prenesemo na grafično kartico v konstantni pomnilniku, nato pa jih prestavimo v deljenega. Zadnjo možnost smo preizkusili zato, ker Mei in dr. [7] trdijo, da je tak način najhitrejši. Avtorji so sicer uporabili drugačen algoritem s 16 niti na blok podatkov, zato je tudi pričakovano, da se rezultati lahko razlikujejo. Za testiranje smo uporabili algoritem Rijndael in 128 MiB podatkov za šifriranje.

Rezultate prikazuje graf na sliki 6.3, iz katerega je razvidno, da je najboljši način tisti, kjer tabele prenesemo v globalni ter jih nato prestavimo v deljeni pomnilnik. Načina, kjer se tabele pri uporabi ne hranijo v deljenem pomnilniku, se izkažeta kot počasnejša, saj je naključni dostop do global-



Slika 6.4: Čaš šifriranja algoritma Rijndael pri različnih lokacijah podključev.

nega in konstantnega počasnejši kot do deljenega. Konstanti pomnilnik je v tem primeru še počasnejši, saj imamo zaradi načina dostopa do tabel veliko zgrešitev v predpomnilniku.

6.1.2.3 Podključi rund

Za hranjenje podključev rund predlagajo Mei [7] in Nishikawa [9] deljeni pomnilnik, medtem ko Li [8] predlaga, da jih hranimo v konstantnem. V naših implementacijah je uporabljen konstanti pomnilnik. Razlog za to je, da vse niti hkrati uporabijo isti podključ znotraj ene runde. Da bi odločitev utemeljili, smo naredili preizkus, kjer hranimo podključe v različnih pomnilnikih. Ponovno je bil uporabljen algoritem Rijndael in 128 MiB podatkov.

Rezultate prikazuje graf na sliki 6.4, iz katerega je razvidno, da je različica, ki hrani podključe v konstantnem pomnilniku najhitrejša. Razlog za to je predpomnjenje podatkov iz konstantnega pomnilnika.

6.1.3 Velikost bloka niti

Ker je velikost substitucijskih tabel v algoritmih večkratnik števila 256, smo izbrali tudi takšno velikost bloka niti. V tem primeru vse niti pri prenašanju

tabel iz globalnega v deljeni pomnilnik sodelujejo hkrati. Poleg tega je pri tej velikosti bloka niti zasedenost multiprocesorja še vedno dovolj visoka, da ne tratimo virov po nepotrebnem.

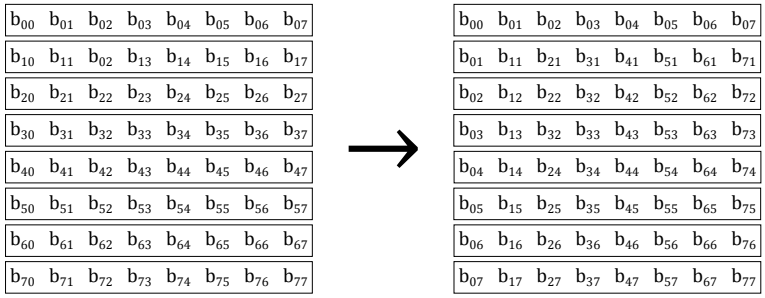
6.2 Vzporedne implementacije z bitnimi rezinami

Alternativen način implementacije bločne šifre je implementacija z bitnimi rezinami. Pri tem načinu si predstavljamo procesor kot SIMD (ang. Single Instruction Multiple Data) računalnik na strojnem nivoju, ki je zmožen naenkrat obdelati toliko podatkov, kolikor je dolžina njegovega registra.

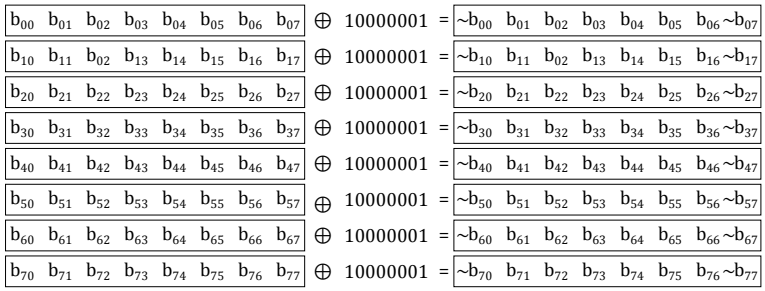
Pri implementacijah z bitnimi rezinami je potrebno podatke najprej predstaviti z bitnimi rezinami oziroma jih transponirati po bitih. To pomeni, da če imamo n m-bitnih števil, jih predstavimo z m n-bitnimi števili, kjer bo j-ti bit i-tega števila predstavljen kot i-ti bit j-tega števila v načinu z bitnimi rezinami. Z drugimi besedami: i-to število bo pri predstavitvi z bitnimi rezinami imelo informacijo o vseh i-tih bitih n-tih števil kot je prikazano na sliki 6.5. V primeru finalistov izbora AES, bo $n = 32$, ker ima CUDA 32-bitne registre in $m = 128$, ker je dolžina bloka enaka 128 bitov.

Za lažjo predstavitev bitnih rezin si bomo pogledali primer na hipotetični bločni šifri z velikostjo bloka 8 bitov, kjer želimo izvesti operacijo XOR med 8 bloki in konstanto $0x81 = 10000001_{bin}$. Običajno je za to potrebno izvesti 8 XOR operacij kot to prikazuje slika 6.6. Kadar pa imamo podatke predstavljene z bitnimi rezinami, pa potrebujemo za isto operacijo izvesti samo negacijo dveh števil kot to prikazuje slika 6.7, saj sta v konstanti $0x81$ samo prvi in osmi bit postavljena na 1. V najslabšem primeru bi bilo potrebno ponovno izvesti osem operacij, kadar bi bila konstanta enaka $0xFF = 11111111_{bin}$.

Takšna vrsta predstavitve podatkov je predvsem učinkovita kadar je večina operacij bitnih kot so OR, AND, XOR in NOT ter rotacije in zamiki, saj jih je enostavno implementirati kot logična vrata. Popolnoma pa so neučinkoviti pri aritmetičnih operacijah, saj bi v tem primeru morali implementirati



Slika 6.5: Predstavitev podatkov z bitnimi rezinami.



Slika 6.6: Za operacijo XOR med 8 števili in konstanto $0x81 = 10000001_{bin}$ je potrebno izvesti osem operacij.

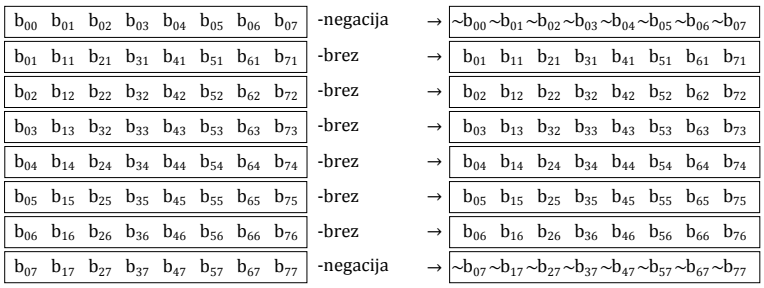
seštevalnike ali množilnike programsko.

Bločne šifra je torej primeren kandidat za implementacijo z bitnimi rezinami kadar vsebuje veliko bitnih operacij. Zato tega sta izmed petih finalistov primerna samo Rijndael in Serpent. Twofish, Mars in RC6 odpadejo, saj vsebujejo veliko aritmetičnih operacij.

6.2.1 Transponiranje podatkov

Preden se lotimo vzporedne implementacije z bitnimi rezinami algoritmov Rijndael in Serpent, bomo razložili kako naredimo vzporedno transponiranje in inverzno transponiranje.

Na računalniku z n-bitnimi registri lahko hkrati obdelamo n blokov v načinu z bitnimi rezinami. Ker uporablja CUDA 32-bitne registre, bo število blokov ene skupine 32. Skupini 32 blokov podatkov bomo rekli sveženj (ang.



Slika 6.7: Za operacijo XOR med 8 števili in konstanto $0x81 = 10000001_{bin}$ pri predstavitvi z bitnimi rezinami je potrebno izvesti samo dve negaciji (samo zadnji in prvi bit sta v število 0x81 postavljena na 1).

bundle).

Naloga algoritma za transponiranje je, da podatke v pomnilniku grafične kartice preoblikuje tako, da so primerni za šifriranja v načinu z bitnimi rezinami. To pomeni, da so podatki v takšni obliki kot da bi jih brali po stolpcih namesto po vrsticah. Ko prenehamo s šifriranjem, pa uporabimo inverzno transponiranje, da jih dobimo nazaj v standardno obliko.

Pri implementaciji smo se želeli izogniti sinhronizaciji med nitmi in atomarnimi operacijami, ki bi povzročile čakanje ene niti na drugo. Zaradi tega smo se odločili izbrati takšno razporeditev podatkov, da bo vsaka nit poskrbela za transponiranje osmih bajtov enega svežnja. Prikaz podatkov, ki jih obdela nit 0 pri transponiranju prikazuje slika 6.8. Veliki indeks predstavlja številko bloka podatkov, mali pa številko bita znotraj bloka. Ker vsebuje en sveženj $32 \times 16B = 512B$ podatkov, ena nit pa obdela $8B$ podatkov, potrebujemo za obdelavo celega svežnja $\frac{512B}{8\frac{B}{nit}} = 64$ niti.

Pri ščepcu za transponiranje podatkov smo uporabili bloke niti velikosti 128. Razlog za takšno število je, da je učinkovitost multiprocesorja višja kot če bi uporabili samo bloke velikosti 64 niti. Bloke smo razdelili na tri dimenzije velikosti po 4, 16 in 2 niti. Takšno velikost prvih dveh dimenzij smo izbrali zaradi lažjega preračunavanja indeksov podatkov, ki jih obdela določena nit znotraj svežnja. Zadnja dimenzija pa je uporabljena zato, da se izbere eden izmed svežnjev, ki ga obdela blok niti.

Ko izberemo ustrezen sveženj za transponiranje na podlagi indeksa bloka niti in tretje dimenzije znotraj bloka, se odmik znotraj svežnja za specifično nit izračuna po sledeči formuli:

$$\texttt{offset}_1 = t_x * 128 + t_y$$
$$\texttt{offset}_2 = t_y * 32 + t_x,$$

kjer t_x predstavlja indeks po prvi dimenziji in t_y po drugi dimenziji.

Bajti za transponiranje se iz vhodnega ščepca za posamezno nit izmed 64-tih izberejo po sledeči formuli:

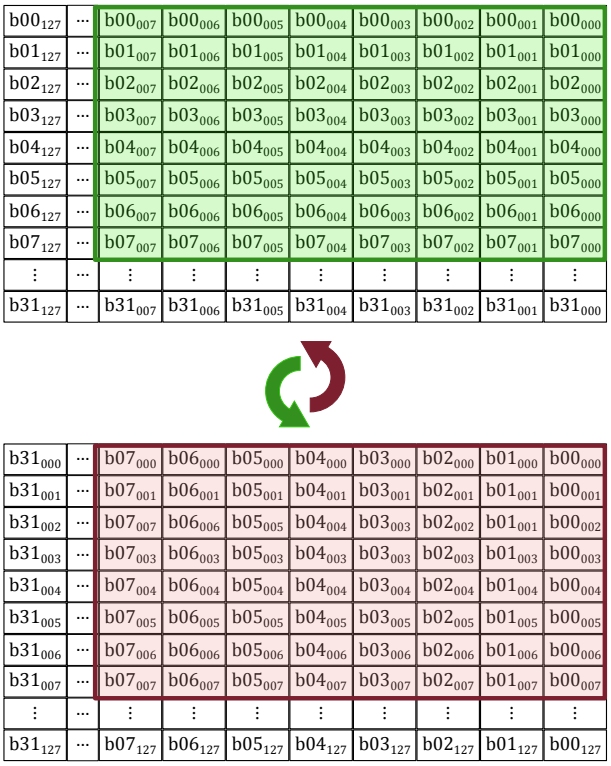
$$\texttt{bin}_i = \texttt{bundle}_{\texttt{in}}[\texttt{offset}_1 + i * 16], i = 0 \ldots 7$$

Sledi zaporedje logičnih operacij s katerimi transponiramo bajte. Najprej izračunamo pozicijo bita, ki ga hočemo izluščiti in shraniti v posamezen izhod:

$$\texttt{bit}_i = (1 << i), i = 0 \ldots 7.$$

Nato izluščimo posamezne bite iz vhodnih bajtov in jih ustrezno zamaknemo.

$$\begin{aligned}
&bout_i = \\
&((\texttt{bin}_0 \ \&\ \texttt{bit}_i) << (0 - i)) \mid \\
&((\texttt{bin}_1 \ \&\ \texttt{bit}_i) << (1 - i)) \mid \\
&((\texttt{bin}_2 \ \&\ \texttt{bit}_i) << (2 - i)) \mid \\
&((\texttt{bin}_3 \ \&\ \texttt{bit}_i) << (3 - i)) \mid \\
&((\texttt{bin}_4 \ \&\ \texttt{bit}_i) << (4 - i)) \mid \\
&((\texttt{bin}_5 \ \&\ \texttt{bit}_i) << (5 - i)) \mid \\
&((\texttt{bin}_6 \ \&\ \texttt{bit}_i) << (6 - i)) \mid \\
&((\texttt{bin}_7 \ \&\ \texttt{bit}_i) << (7 - i)) \\
&i = 0 \ldots 7.
\end{aligned}$$



Slika 6.8: Podatki, ki jih obdeluje nit 0 pri navadnem in inverznem transponiranju.

Kadar je število zamika negativno število, se zamakne v nasprotno smer. Na koncu še shranimo izhod po sledeči formuli:

$$\texttt{bundle}_{\texttt{out}}[\texttt{offset}_2 + i * 4] = \texttt{bout}_i, i = 0 \ldots 7.$$

Inverzno transponiranje se izvede v obratnem vrstnem redu. V tem primeru zamenjamo tudi velikost prvih dveh dimenzij bloka niti.

6.2.2 Ustvarjanje števcev in operacija XOR med tokom ključev in čistopisom

Ker za šifriranje več blokov uporabljamo način CTR, moramo pravzaprav šifrirati števce. Zaradi tega moramo imeti pred transponiranjem že ustvar-

jene števce, ki jih bomo transponirali, šifrirali ter na koncu še inverzno transponirali. Pred samim transponiranjem zato poženemo še en ščepec, ki skrbi za ustrezno ustvarjanje števcev na podoben način kot smo to počeli pri vzporednih algoritmih z razporejanjem podatkov.

Poleg tega je del načina CTR tudi operacija XOR med šifriranimi števci in čistopisom, da dobimo šifropis. Zato po inverznem transponiranju poženemo še ščepec, ki poskrbi za operacijo XOR. Podatke smo razdelili med niti tako, da vsaka nit poskrbi za XOR enega bloka podatkov s svojim šifriranim števcem.

Skupaj imamo tako za celotno šifriranje in obdelavo podatkov pet ščepcev: Ustvarjanje števcev, transponiranje, šifriranje z bitnimi rezinami, inverzno transponiranje in operacija XOR med šifriranimi števci in čistopisom.

6.2.3 Vzporedna implementacija algoritma Rijndael z bitnimi rezinami

6.2.3.1 Delitev podatkov med niti

Pri algoritmih na osnovi razporejanja podatkov smo izbrali pristop, kjer vsaka nit obdela svoj blok podatkov. Za to smo potrebovali štiri registre.

Pri implementacijah z bitnimi rezinami pa nastane težava, saj imamo 32 zaporednih blokov podatkov shranjenih v 128 besedah, za kar bi potrebovali 128 registrov. To je občutno preveč registrov na eno nit, zato smo morali izbrati drugačen način delitve podatkov.

Odločili smo se za takšen pristop, kjer vsaka nit obdela 8 registrov svežnja, kar je isto kot osem skupin bitov. Takšno delitev smo izbrali, ker potrebujemo za izračun operacija `SubBytes` cel bajt navadnega podatkovnega bloka.

Pri tej vrsti delitve podatkov potrebujemo 16 niti na en blok, saj je običajen blok podatkov sestavljen iz 16 bajtov. Skupino 8 besed, ki jih obdeluje ena nit, bomo v nadaljevanju imenovali delovni blok.

Ponovno smo izbrali blok niti velikosti 128 niti in se zaradi lažjega indeksiranja odločili za dimenzije velikosti 4, 4 in 8. Zadnja dimenzija se zopet

uporablja za izbiro ustreznega svežnja, ki ga obdeluje blok niti.

6.2.3.2 SubBytes

Kot smo že omenili pri opisu algoritma Rijndael, je operacija `SubBytes` sestavljena iz računanja inverznega elementa operacije množenja v končnem obsegu $GF(2^8)$ in afine transformacije. Pri navadni implementaciji algoritma Rijndael imamo ti dve operaciji združeni v eno substitucijsko tabelo oziroma štiri tabele, če ju združimo še z `ShiftRows` in `MixColumns`.

Ker pa imamo podatke predstavljene v načinu z bitnimi rezinami, ne moremo uporabiti poizvedb v tabele, saj bi morali za pridobitev vrednosti znova transponirati podatke nazaj. Zato moramo najti drugačen način, kako implementirati operacijo `SubBytes`.

Afino transformacijo lahko implementiramo brez težav kot zaporedje logičnih operacij med registri, saj gre za preprosto linearno operacijo. Večjo težavo nam predstavlja računanje inverznega elementa.

Obstaja več načinov kako izračunati inverzni element. Mi smo izbrali pristop, ki ga je predstavil Rijmen [27]. V svojem delu je raziskoval, kako učinkovito implementirati substitucijsko tabelo na strojnem nivoju z logičnimi operacijami, kar nam pride prav pri naši implementaciji z bitnimi rezinami.

Njegovo idejo je izboljšal še Canright [28], čigar pristop uporabljajo tudi Reberio in dr. [14] pri svoji implementaciji z bitnimi rezinami algoritma Rijndael za CPE.

Ker je razlaga celotnega postopka zapletena, bomo prikazali samo njegovo idejo. Vsak element v končnem obsegu $GF(2^8)$ se da predstaviti kot polinom stopnje 7 s koeficienti v $GF(2)$ in množenjem po modulu nedeljivega polinoma stopnje 8:

$$p(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0.$$

Ta isti element lahko napišemo tudi kot polinom prve stopnje s koeficienti v $GF(2^4)$ in množenjem po modulu nedeljivega polinoma druge stopnje:

$q(x) = x^2 + Ax + B$, ki ima koeficiente v $GF(2^4)$:

$$p(x) = bx + c.$$

Računanje inverza se s tem poenostavi na:

$$(bx + c)^{-1} = b(b^2B + bcA + c^2)^{-1}x + (c + bA)(b^2B + bcA + c^2)^{-1}.$$

Canright je naredil še korak dlje, tako da je koeficiente v $GF(2^4)$ predstavil kot polinome prve stopnje s koeficienti v $GF(2^2)$. Koeficiente v $GF(2^2)$ pa je predstavil kot polinome prve stopnje s koeficienti v $GF(2)$, kjer imamo samo še dva elementa: 0 in 1. Na tem nivoju je iskanje inverza sestavljeno iz kombinacije logičnih operacij, zato je idealno za našo implementacijo.

Ko končamo z računanjem, moramo delovni blok shraniti v deljeni pomnilnik, da bo dostopen še ostalim nitim pri operacijah `ShiftRows` in `MixColumns`.

Ker pa je delovni blok velik 8 registrov, zavzame 8 pomnilniških bank, kar pa je delitelj števila 32. Zaradi tega imamo v deljenem pomnilniku konflikte. To lahko preverimo tako, da pogledamo naslove začetnih bank delovnih blokov, ki jih shranijo niti v deljeni pomnilnik. Te lahko izračunamo po sledeči formuli:

$$\texttt{addr}_i \equiv 8i \pmod{32},\ i = 0 \ldots 31.$$

Če sedaj zapišemo indeks i z dvema drugima indeksoma j in k:

$$i = 4j + k,\ j = 0 \ldots 7,\ k = 0 \ldots 3,$$

se nam izračun poenostavi:

$$\begin{aligned}
&\texttt{addr}_i \equiv 8i \pmod{32},\ i = 0 \ldots 32 \\
&\texttt{addr}_{4j+k} \equiv 8(4j+k) \pmod{32},\ j = 0 \ldots 7,\ k = 0 \ldots 3 \\
&\texttt{addr}_{4j+k} \equiv 32j + 8k \pmod{32},\ j = 0 \ldots 7,\ k = 0 \ldots 3 \\
&\texttt{addr}_{4j+k} \equiv 8k \pmod{32},\ j = 0 \ldots 7,\ k = 0 \ldots 3.
\end{aligned}$$

Iz računa vidimo, da skupine po 8 niti pišejo v iste banke, zato imamo osemsmerni konflikt.

Temu se lahko izognemo tako, da dodamo delovnemu bloku še eno lažno (ang. dummy) besedo in s tem zasedemo 9 bank. Račun za naslov začetnega naslova banke se v tem primeru izračuna kot:

$$\texttt{addr}_i \equiv 9i \pmod{32},\ i = 0 \ldots 31.$$

Ker 9 ni delitelj števila 32 se začetni naslovi bank vseh niti razlikujejo.

6.2.3.3 ShiftRows in MixColumns

Operaciji `ShiftRows` in `MixColumns` smo združili pod eno operacijo. Da bi lažje razumeli kako izračunamo te dve operacije v načinu z bitnimi rezinami, si še enkrat poglejmo združeno formulo `ShiftRows` in `MixColumns`:

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} S[a_{0,0}] & S[a_{0,1}] & S[a_{0,2}] & S[a_{0,3}] \\ S[a_{1,1}] & S[a_{1,2}] & S[a_{1,3}] & S[a_{1,0}] \\ S[a_{2,2}] & S[a_{2,3}] & S[a_{2,0}] & S[a_{2,1}] \\ S[a_{3,3}] & S[a_{3,0}] & S[a_{3,1}g & S[a_{3,2}] \end{bmatrix}.$$

Če jo razpišemo kot 16 linearnih enačb in malenkost preuredimo člene, dobimo:

$$\begin{aligned}
a_{0,0} &= 2S[a_{0,0}] \oplus 3S[a_{1,1}] \oplus S[a_{2,2}] \oplus S[a_{3,3}] \\
a_{0,1} &= 2S[a_{0,1}] \oplus 3S[a_{1,2}] \oplus S[a_{2,3}] \oplus S[a_{3,0}] \\
&\vdots \\
a_{3,2} &= 2S[a_{3,1}] \oplus 3S[a_{0,2}] \oplus S[a_{1,3}] \oplus S[a_{2,0}] \\
a_{3,3} &= 2S[a_{3,2}] \oplus 3S[a_{1,2}] \oplus S[a_{2,3}] \oplus S[a_{3,0}]
\end{aligned}$$

Ko pogledamo indekse pri posameznih členih enačbe, vidimo da se vzorec ponavlja. Indeksi členov enačbe $a_{i,j} = 2S[a_{idx_0}] \oplus 3S[a_{idx_1}] \oplus S[a_{idx_2}] \oplus S[a_{idx_3}]$ so izračunani po sledeči formuli:

$$\begin{aligned}
\texttt{idx}_0 &= (i+0 \mod 4,\ i+j+0 \mod 4) \\
\texttt{idx}_1 &= (i+1 \mod 4,\ i+j+1 \mod 4) \\
\texttt{idx}_2 &= (i+2 \mod 4,\ i+j+2 \mod 4) \\
\texttt{idx}_3 &= (i+3 \mod 4,\ i+j+3 \mod 4).
\end{aligned}$$

Vsaka nit skrbi za en delovni blok, kar predstavlja en bajt znotraj posameznega bloka podatkov. Zaradi tega moramo prebrati štiri delovne bloke iz deljenega pomnilnika, jih ustrezno pomnožiti in sešteti po modulu dva.

Množenje z 2, oziroma polinomom $p_2(x) = x$ po modulu $q(x) = x^8 + x^4 + x^3 + x + 1$, je v navadnih implementacijah z razporejanjem podatkov implementirano kot rotiranje bajta in pogojni XOR z $0x11B$, če je zgornji bit enak 1. Množenje s polinomom $p(x) = x$ namreč prestavi koeficiente polinoma za eno mesto naprej. Na koncu pa je treba še deliti s polinomom $q(x)$ in vzeti ostanek, če je stopnja polinoma večja kot sedem.

Množenje s polinomom $p(x) = x$ smo zaradi tega implementirali kot zamenjavo registrov znotraj delovnega bloka. Stari register r_0, ki vsebuje najmanj pomembne bite, postane novi r_1, stari r_1 postane novi r_2, itd. Sledijo še tri

XOR operacije s staro vrednostjo registra r_7, ki se sedaj nahaja v r_0:

$$r_1 = r_1 \oplus r_0$$
$$r_3 = r_3 \oplus r_0$$
$$r_4 = r_4 \oplus r_0$$

Množenje s 3 oziroma polinomom $p_3(x) = x + 1$ smo implementirali kot operacijo XOR med delovnim blokom ter produktom delovnega bloka s polinomom $p_2(x) = x$, saj velja $p_3(x) = p_2(x) + 1$.

6.2.3.4 AddRoundKey

Operacijo AddRoundKey smo implementirali podobno kot prikazuje slika pri predstavitvi bitnih rezin 6.2. Vsaka nit vzame ustrezen bajt podključa runde in pogleda, kateri biti so nastavljeni na 1. Če je bit b_i nastavljen na 1, potem negiramo register r_i.

6.2.4 Vzporedna implementacija algoritma Serpent z bitnimi rezinami

Pri implementaciji z bitnimi rezinami algoritma Serpent smo imeli manj težav, saj je Serpent že v osnovi načrtovan tako, da ga lahko enostavno implementiramo z bitnimi rezinami. Največjo težavo nam je zopet predstavljalo število registrov na posamezno nit.

Navadna implementacija algoritma Serpent ima blok podatkov shranjen v štirih besedah oziroma registrih. Za izračun tabel in ostalih operacij potrebuje nit vse štiri registre oziroma en bit posameznega registra za izračun izhoda istega bita.

Zaradi tega smo izbrali takšen delovni blok, da razpolaga vsaka nit z enim bitom posamezne besede v načinu z bitnimi rezinami. Število podatkov na eno nit je v tem primeru zopet 16 bajtov oziroma štiri besede, saj moramo vzeti cel sveženj 32 podatkovnih blokov. Število niti na en sveženj je v tem

primeru $\frac{512B}{\frac{16B}{nit}} = 32$ niti.

Drugi problem, ki smo ga imeli je bilo rotiranje besed pri linearni transformaciji. Ker vsaka nit razpolaga samo z enim bitom neke besede predstavljene v navadnem načinu, smo rabili neke vrste komunikacijo med ostalimi nitmi, ki imajo shranjene druge bite. To smo naredili s pomočjo deljenega pomnilnika.

6.2.5 Časovna zahtevnost implementacij z bitnimi rezinami

Šifriranje podatkov pri obeh implementacijah z bitnimi rezinami je sestavljeno iz računanja podključev, ustvarjanja števcev, transponiranja, šifriranja, inverznega transponiranja ter operacije XOR med šifriranimi števci in čistopisom.

Za izračun časovne zahtevnosti bomo zopet napisali odvisnosti glede na število vhodnih blokov n pri številu procesorjev oziroma niti P. Izračun podključev je konstantna operacija C_1. Računanje števca je konstantna operacija, ki jo moramo ponoviti tolikokrat, kolikor imamo blokov. Če imamo P niti, se delo razdeli med njimi. Časovna zahtevnost te operacije je v tem primeru enaka: $T_P^{(ctr)}(n) = \frac{n}{P}C_2$. Podobno časovno zahtevnost ima tudi operacija XOR, le da z drugo konstanto C_3.

En sveženj je sestavljen iz 32 blokov podatkov, za katerega pri transponiranju poskrbi 64 niti. Če pogledamo, koliko niti potrebujemo dejansko za obdelavo enega bloka, vidimo da je količnik enak $\frac{64}{32} = 2$. Za obdelavo enega bloka podatkov porabimo torej 2 niti. Časovna zahtevnost operacije transponiranja in inverznega transponiranja je zaradi tega enaka:

$$T_P^{(trans)}(n) = \frac{2n}{P}C_4$$

Do podobnega izračuna pridemo tudi pri računanju časovne zahtevnosti šifriranja. Sveženj, ki je sestavljen iz 32 podatkovnih blokov obdela 16 niti pri algoritmu Rijndael. Število niti na en blok je v tem primeru enako $\frac{16}{32} = \frac{1}{2}$. V primeru algoritma Serpent pa je število niti na en blok enako $\frac{32}{32} = 1$.

Časovna zahtevnost šifriranja je v tem primeru enaka

$$T_P^{(rijndael\_cipher)} = \frac{n}{2P}C_6$$

za algoritem Rijndael in

$$T_P^{(serpent\_cipher)} = \frac{n}{P}C_7$$

za algoritem Serpent.

Celotno časovno zahtevnost algoritma Rijndael lahko torej zapišemo kot:

$$T_P^{(rijndael\_total)}(n) = C_1 + \frac{n}{P}C_2 + \frac{2n}{P}C_4 + \frac{n}{2P}C_6 + \frac{2n}{P}C_5 + \frac{n}{P}C_3.$$

Če definiramo nove konstante $2C_4 = C_8$, $\frac{C_6}{2} = C_9$ in $2C_5 = C_{10}$, dobimo:

$$T_P^{(rijndael\_total)}(n) = C_1 + \frac{n}{P}(C_2 + C_8 + C_9 + C_{10}) = C_1 + \frac{n}{P}C_{11} = \theta(n).$$

Teoretično pohitritev, če je operacija razširjanja ključa hitra operacija, lahko torej izračunamo kot:

$$\frac{T_1^{(rijndael)}(n)}{T_P^{(rijndael\_total)}} = \frac{C_1 + nC_{12}}{C_1 + \frac{n}{P}C_{11}} \approx \frac{nC_{12}}{\frac{n}{P}C_{11}} \approx PC_{13}$$

Do podobnega izračuna pridemo tudi pri algoritmu Serpent.

Poglavje 7

Primerjava implementacij

V tem poglavju bomo primerjali zaporedne in vzporedne implementacije vseh petih finalistov AES. Najprej bomo predstavili testne podatke in kako smo jih ustvarili. Sledi opis, kako smo testirali. Na koncu je še prikaz rezultatov ter njihova razlaga.

7.1 Podatki

Vhodne datoteke, ki smo jih uporabili za testiranje smo ustvarili tako, da smo naredili naključne nize različnih dolžin z enako verjetnostjo pojavitve vsakega znaka. Testirali smo s 24 različnimi dolžinami.

Dolžine nizov smo določili tako, da smo začeli pri 16 B. Velikosti smo nato množili z 2, dokler nismo prišli do velikosti 128 MiB. Takšen pristop smo izbrali zato, da se opazijo trendi pri majhnih velikostih. Poleg tega pa vidimo, kako se algoritmi obnesejo pri velikih podatkih. Podoben pristop uporabljajo tudi sorodni članki.

Ključ je bil dolžine 128-bitov in je bil prav tako ustvarjen naključno. Zaradi majhne časovne zahtevnosti in neodvisnosti od bloka podatkov, njegove razširitve nismo prišteli k celotnemu času šifriranja.

Proizvajalec	Toshiba
Model	Satellite L750-1NJ
CPE	Intel Core i7-2670QM
Pomnilnik	DDR3, PC3-10700, 8GB
OS	Windows 8.1 Pro 64-bit
Prevajalnik	Visual C++ 12.0
GPE	NVIDIA GeForce GT 525M
CUDA Toolkit	v6.5
Arhitektura CUDA	Fermi
Računska zmogljivost	2.1
Hitrost grafičnega jedra	475 MHz
Število multiprocesorjev	2
Število CUDA jeder	96
Hitrost CUDA jedra	900 MHz
Pomnilnik	2048 MB
Hitrost spomina	900 MHz

Tabela 7.1: Testni računalnik

7.2 Testiranje

Testno okolje v katerem smo poganjali naše preizkuse in podrobnejša specifikacija grafične kartice je prikazana v tabeli 7.1.

Testirali smo tako, da smo 1000 krat pognali šifriranje datoteke neke velikosti ter izračunali povprečje. Za ta namen smo uporabili Python skripto, ki je skrbela za poganjanje programa in računanje povprečja. Znotraj vsakega preizkusa smo merili čase različnih operacij. Za ta namen smo uporabili C++ knjižnico Boost Chrono, ki zagotavlja natančne meritve ne glede na operacijski sistem. Pri CPE verzijah smo merili samo čas šifriranja. Pri CUDA in OpenCL pa smo k času šifriranja prišteli tudi čas prenosa podatkov. Pri vzporednih verzijah namenoma nismo prišteli še časov za rezervacijo prostora in prevajanje programov pri OpenCL, saj lahko predpostavljamo, da imamo program že preveden in prostor rezerviran ob zagonu.

Časa dešifriranja nismo upoštevali, saj je dešifriranje v načinu CTR enako šifriranju.

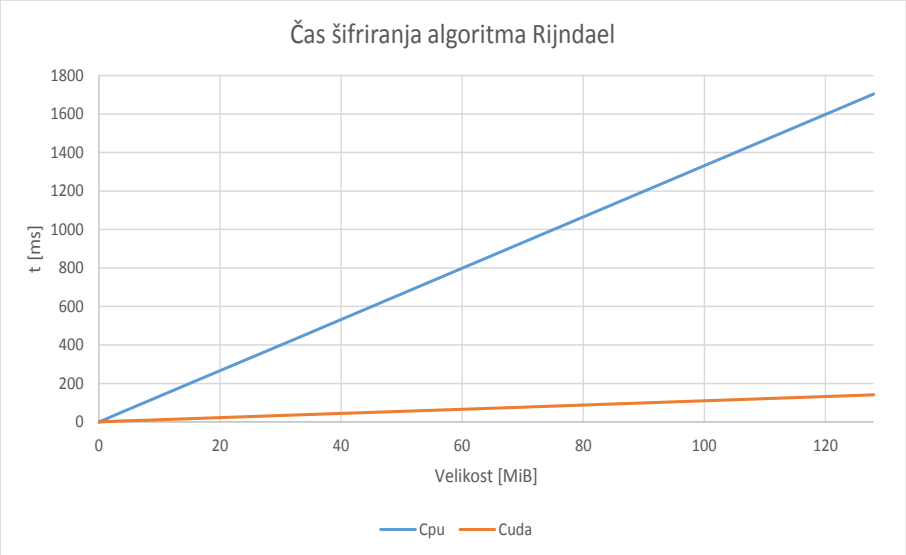
7.3 Rezultati in njihova razlaga

To podpoglavje je razdeljeno na štiri dele. V prvem delu so narejene primerjave med zaporednimi in vzporednimi CUDA implementacijami na osnovi razporejanja podatkov. V drugem delu primerjamo CUDA in OpenCL implementacij na osnovi razporejanja podatkov ter v tretjem delu še CUDA implementacije na osnovi razporejanja podatkov in implementacije z bitnimi rezinami. Na koncu poglavja je še strnjen opis rezultatov.

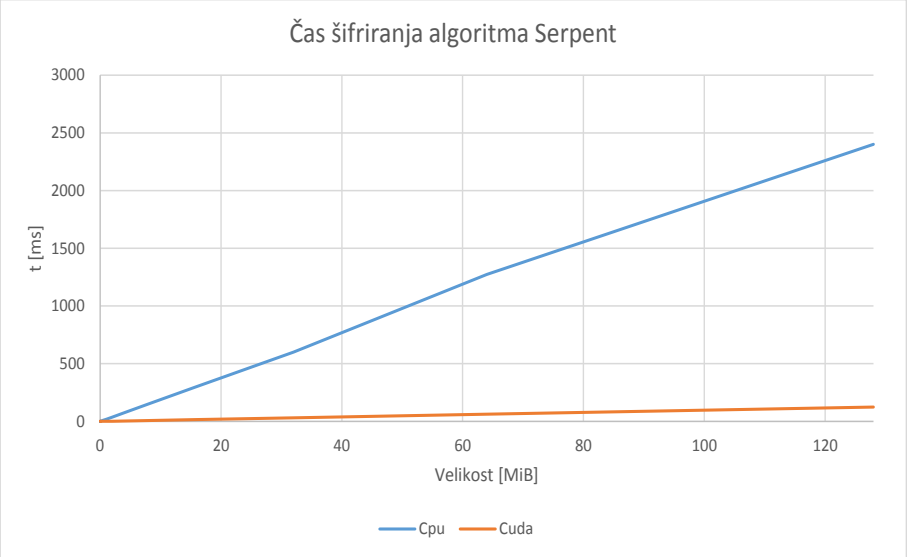
7.3.1 Primerjava zaporednih CPE implementacij in CUDA implementacij na osnovi razporejanja podatkov

7.3.1.1 Časovna zahtevnost

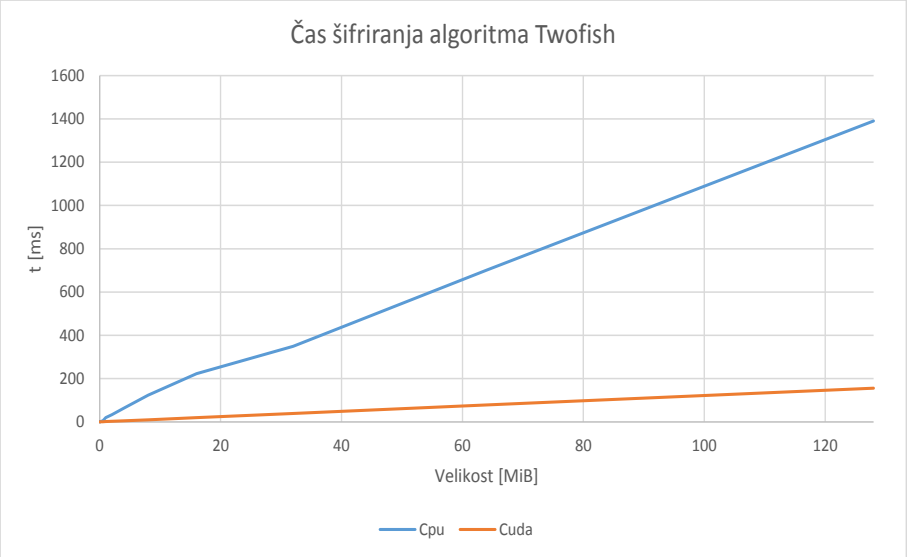
Odvisnost časa šifriranja od velikosti vhodnih podatkov, za vseh pet algoritmov, prikazujejo grafi na slikah 7.1, 7.2, 7.3, 7.5 in 7.4.



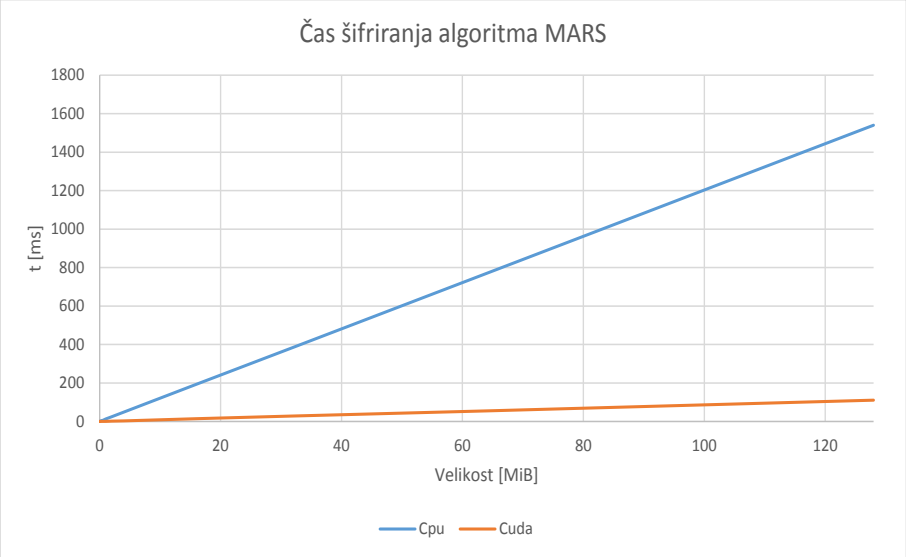
Slika 7.1: Čaš šifriranja algoritma Rijndael.



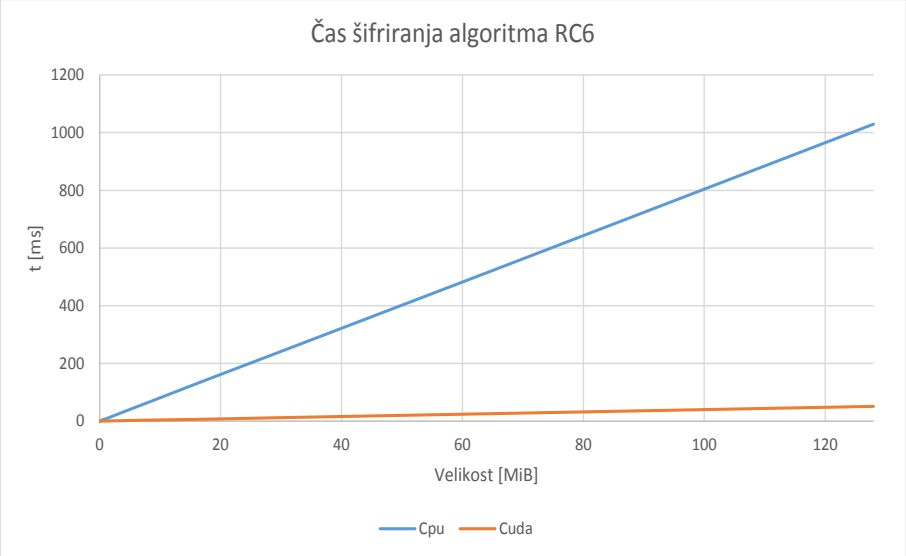
Slika 7.2: Čaš šifriranja algoritma Serpent.



Slika 7.3: Čaš šifriranja algoritma Twofish.



Slika 7.4: Čaš šifriranja algoritma MARS.



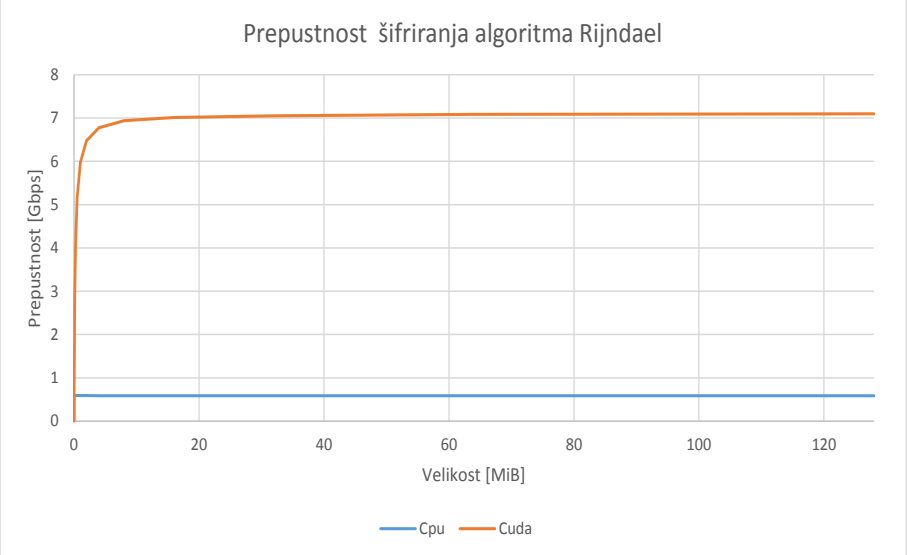
Slika 7.5: Čaš šifriranja algoritma RC6.

Iz grafov je razvidno, da je časovna zahtevnost tako zaporednih kot tudi vzporednih implementacij linearna z različno hitrostjo naraščanja. Takšen

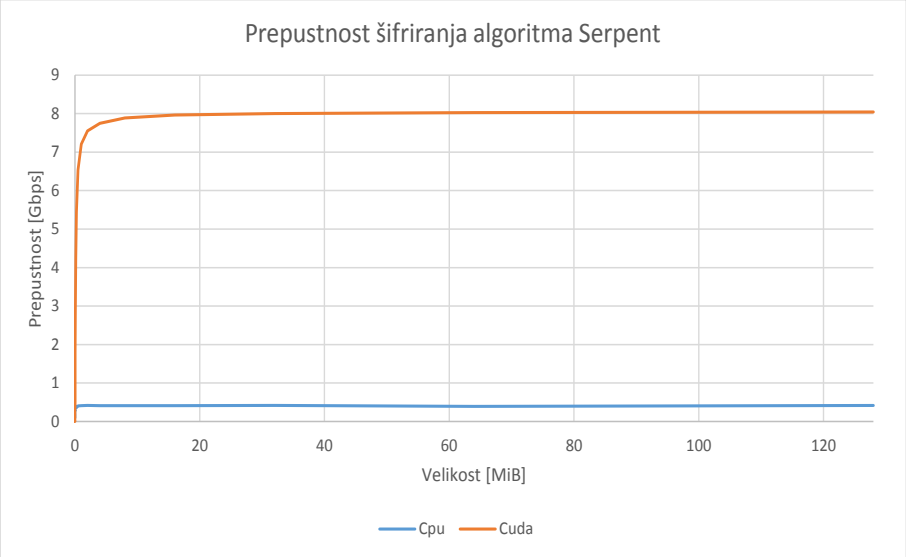
rezultat je pričakovan, saj smo do podobnih ugotovitev prišli tudi pri teoretičnem izračunu časovne zahtevnosti.

7.3.1.2 Prepustnost

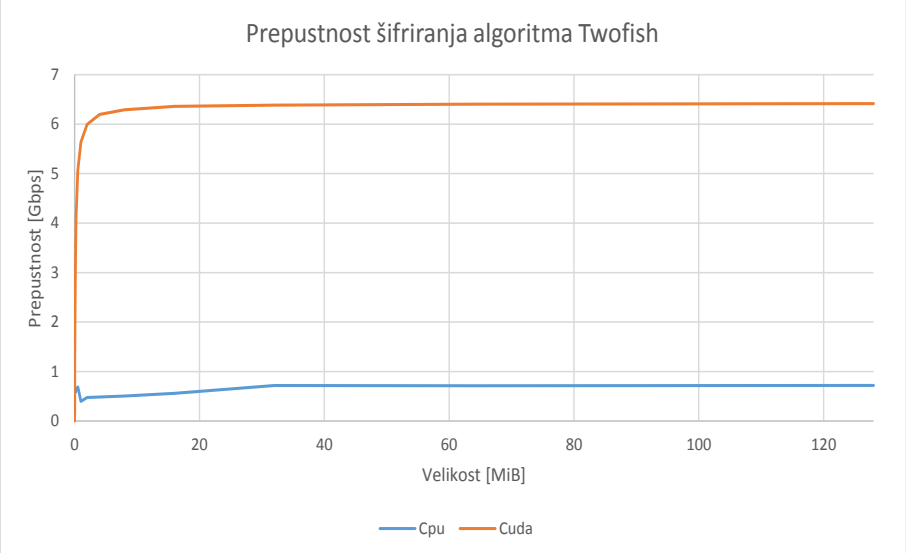
Prepustnost je količina, ki nam pove, koliko podatkov uspemo obdelati na časovno enoto. Odvisnost prepustnosti od velikosti vhodnih podatkov, za vseh pet algoritmov, prikazujejo grafi na slikah 7.6, 7.7, 7.8, 7.10 in 7.9.



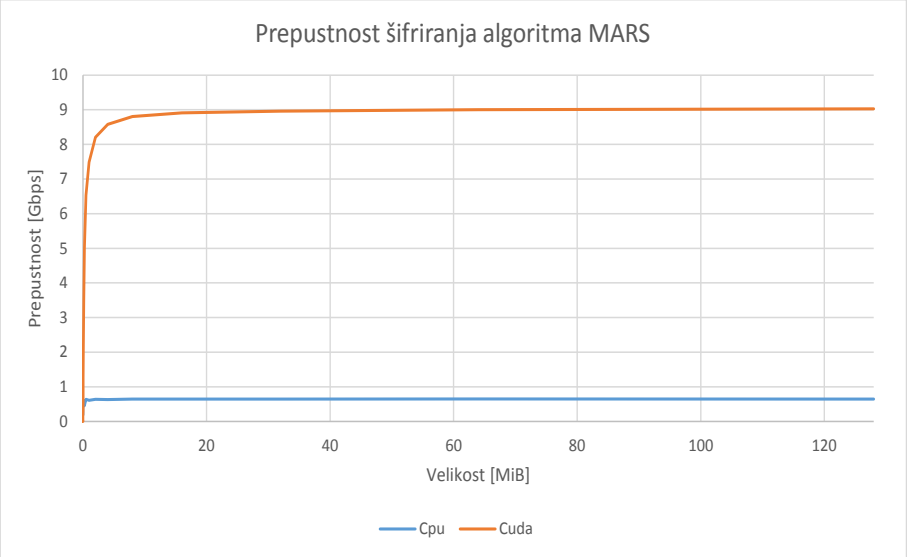
Slika 7.6: Prepustnost šifriranja algoritma Rijndael.



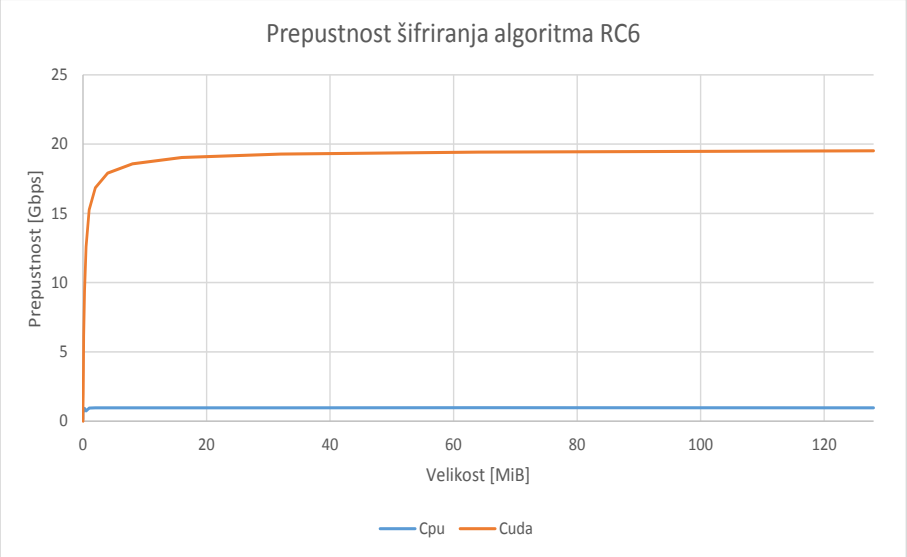
Slika 7.7: Prepustnost šifriranja algoritma Serpent.



Slika 7.8: Prepustnost šifriranja algoritma Twofish.



Slika 7.9: Prepustnost šifriranja algoritma MARS.

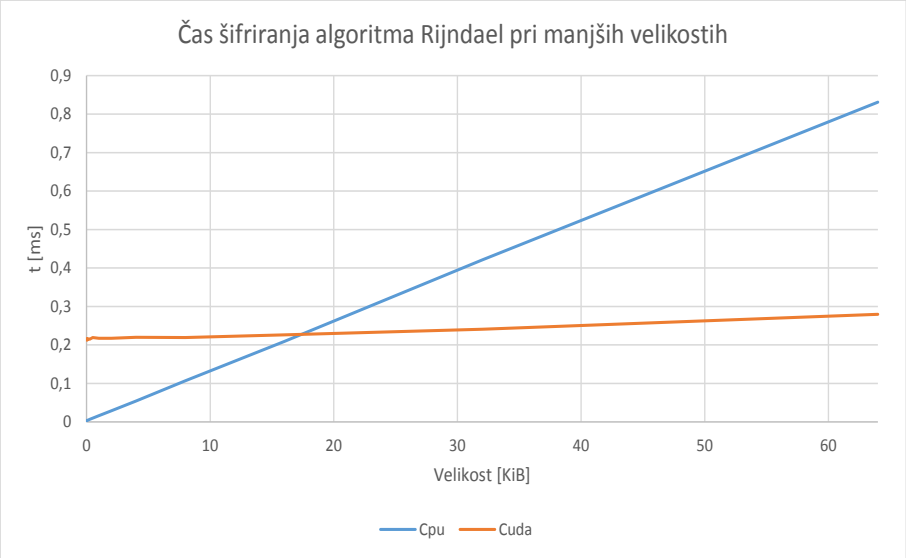


Slika 7.10: Prepustnost šifriranja algoritma RC6.

Na grafih je zanimivo to, da prepustnost vzporednih verzij najprej močno narašča, nato pa se ustavi ter do konca skoraj ne naraste več. Na drugi strani je prepustnost zaporednih različic skoraj celoten čas konstantna. To

si razlagamo tako, da pride pri določeni velikosti podatkov do zapolnitve sistema. Takrat niti ne morejo več obdelati vseh blokov naenkrat vzporedno. Zaradi tega morajo preostali bloki počakati v vrsti na kasnejšo obdelavo.

Posledica tega je, da je časovna zahtevnost vzporednih algoritmov do določene velikosti vhodnih blokov približno konstantna. Ko pa presežemo to vrednost, pa postane linearna. To še dodatno dokazuje graf na sliki 7.11, ki prikazuje čas šifriranja algoritma Rijndael pri manjših velikostih vhodnih podatkov. Razvidno je, da je čas vzporedne implementacije približno konstanten, medtem ko čas zaporedne implementacije linearno narašča. Na začetku grafa je vzporedna različica sicer počasnejša od zaporedne, kar je posledica režijskih stroškov pri zagonu CUDA programa.



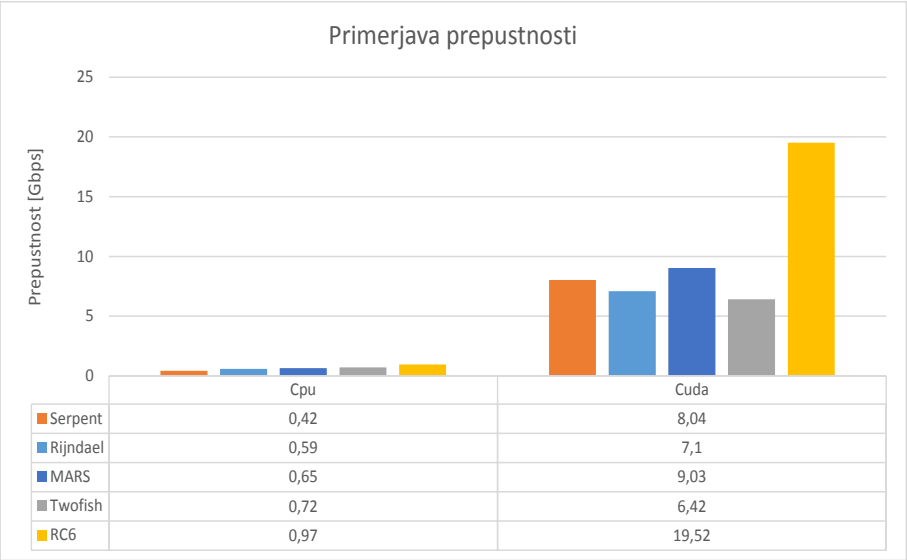
Slika 7.11: Čaš šifriranja algoritma Rijndael pri majhnih velikostih vhodnih datotek.

7.3.1.3 Pohitritev

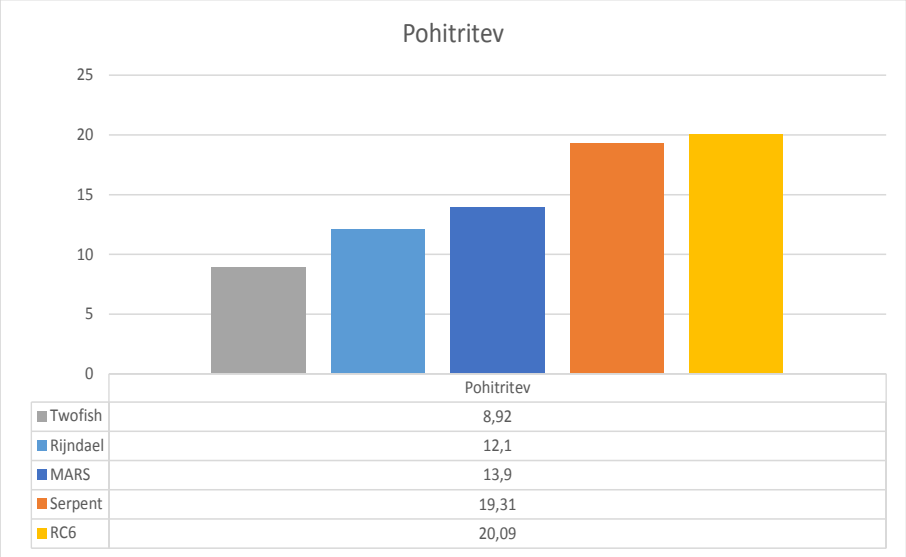
Pohitritev je količnik med časom šifriranja zaporedne in vzporedne implementacije. Lahko ga izračunamo tudi kot količnik med prepustnostjo vzporedne in zaporedne implementacije.

Če primerjamo prepustnost pri 128 MiB podatkov, lahko dobimo občutek kateri izmed algoritmov doseže najvišjo pohitritev 7.12. Iz grafa je razvidno, da se vrstni red najhitrejših algoritmov razlikuje med zaporednimi in vzporednimi različicami. Tukaj predvsem izstopa algoritem Serpent, ki je med zaporednimi implementacijami najpočasnejši, saj je sestavljen iz največ rund.

Našo ugotovitev potrjuje tudi graf primerjave pohitritve na sliki 7.13. Iz njega je razvidno, da dosežemo najvišjo pohitritev pri algoritmu RC6. Visoko pohitritev dosežemo tudi pri algoritmu Serpent, nekoliko slabšo pri algoritmu MARS in Rijndael ter najslabšo pri algoritmu Twofish.



Slika 7.12: Primerjava prepustnosti zaporednih in vzporednih implementacij pri velikosti 128 MiB.



Slika 7.13: Primerjava pohitritve algoritmov pri velikosti podatkov 128 MiB.

Da bi razumeli, zakaj dosežemo pri prvih dveh algoritmih toliko višjo pohitritev, se moramo vprašati kje se ti dve implementaciji razlikujeta od ostalih. Do tega pride zato, ker algoritma Serpent in RC6 ne uporabljata tabel shranjenih v pomnilniku kot ostali algoritmi. Zaradi tega imajo slednji veliko poizvedb v deljeni pomnilnik in posledično tudi konfliktov zaradi naključnih dostopov. Poleg tega je potrebno naprej tabele prenesti iz globalnega v deljeni pomnilnik, kar še dodatno upočasnjuje šifriranje. Arhitektura CUDA je namreč najbolj učinkovita takrat, kadar je delež aritmetično-logičnih operacij v primerjavi s poizvedbami v pomnilnik mnogo večji.

Našo ugotovitev dokazuje tudi tabela 7.2 v kateri imamo prikazano število konfliktov na en snop pri velikosti vhodnih podatkov 128 MiB. Iz nje je razvidno, da imata RC6 in Serpent nič konfliktov, medtem ko ima največ konfliktov algoritem Twofish.

	Rijndael	**Serpent**	**Twofish**	**MARS**	**RC6**
konflikti/snop	287,33	0	387,92	106,33	0

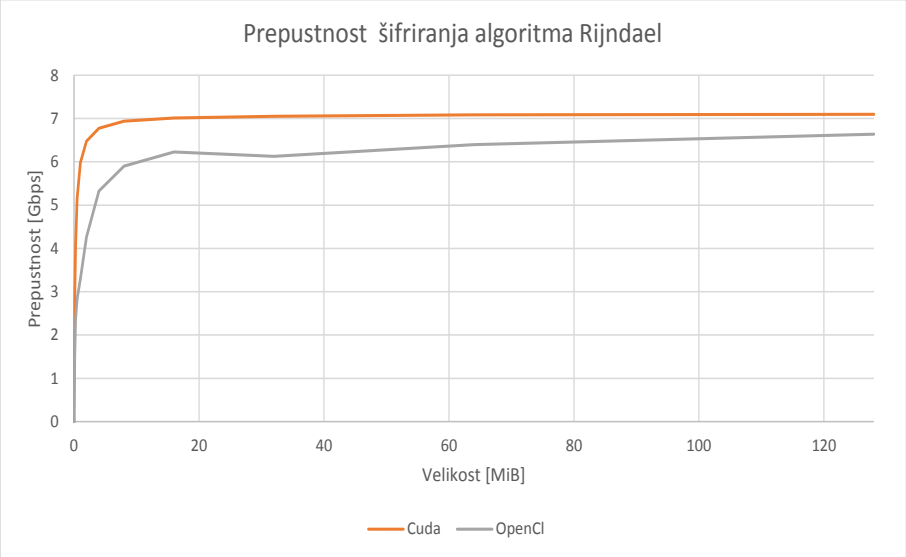
Tabela 7.2: Število zahtev in konfliktov v deljenem spominu.

7.3.2 Primerjava CUDA in OpenCL implementacij vzporednih algoritmov z razporejanjem podatkov

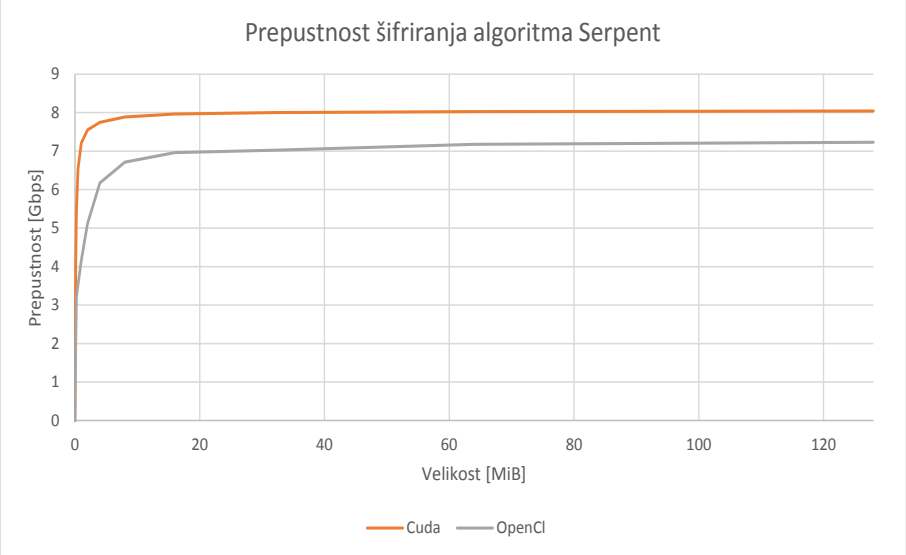
Pri primerjavi implementacij CUDA in OpenCL smo naleteli na nekaj ovir, saj je kljub temu, da obstajajo preslikave med večino sistemskih klicev, še vedno nekaj takih, ki se razlikujejo. Poleg tega se čas izvedbe sorodnih klicev lahko razlikuje med CUDO in OpenCL.

Prav tako smo se morali vprašati ali bomo pri implementacijah OpenCL merili tudi čas za izbiro kontekstna in naprave, čas za branje programske kode ščepca, njegovo prevajanje v PTX kodo in na koncu v strojno kodo. Na koncu smo se odločili, da bomo primerjali samo čas prenosa podatkov in šifriranje za obe vrsti implementacij. Razlog za to je, da lahko predpostavimo, da uporabnik vzpostavi okolje, prevede programsko kodo ščepca itd, že ob zagonu nekega progama za šifriranja. Podobno tudi nismo merili časa vzpostavitve grafične kartice pri implementacijah napisanih v CUDI. To nam bo tudi pokazalo ali se sama izvedba jedra razlikuje glede na platformo.

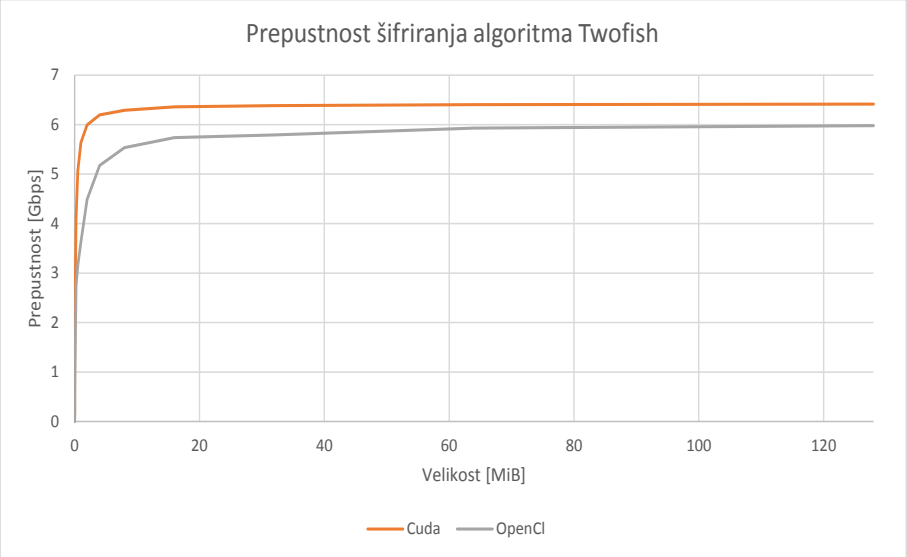
Primerjave prepustnosti implementacij CUDA in OpenCL prikazujejo grafi na slikah 7.14, 7.15, 7.16, 7.18 in 7.17. Iz njih se da razbrati, da so implementacije OpenCL v vseh primerih počasnejše od implementacij CUDA.



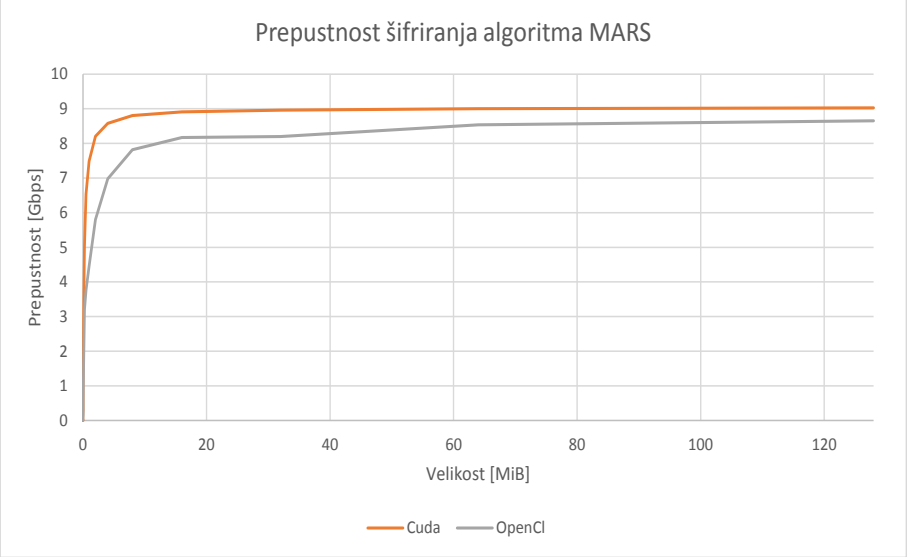
Slika 7.14: Primerjava prepustnosti implementacij CUDA in OpenCL algoritma Rijndael.



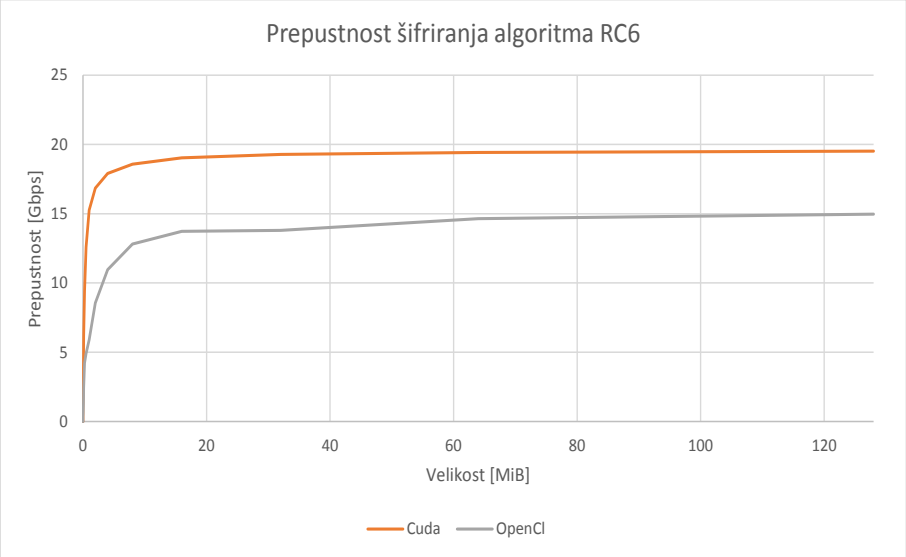
Slika 7.15: Primerjava prepustnosti implementacij CUDA in OpenCL algoritma Serpent.



Slika 7.16: Primerjava prepustnosti implementacij CUDA in OpenCL algoritma Twofish.



Slika 7.17: Primerjava prepustnosti implementacij CUDA in OpenCL algoritma MARS.



Slika 7.18: Primerjava prepustnosti implementacij CUDA in OpenCL algoritma RC6.

Razliko med prepustnostjo implementacij CUDA in OpenCL prikazuje tudi tabela 7.3. Vidimo, da je najvišja razlika v prepustnosti pri algoritmu RC6, medtem ko je ta pri algoritmu Rijndael najmanjša.

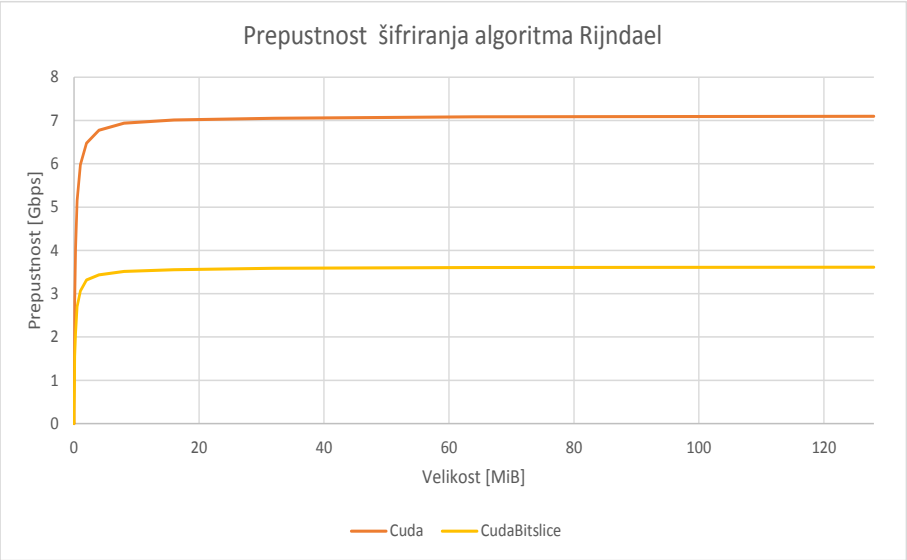
	Rijndael	**Serpent**	**Twofish**	**RC6**	**MARS**
Razlika [%]	1,07	11,23	7,28	30,39	4,32

Tabela 7.3: Razlika v prepustnosti med implementacijami CUDA in OpenCL.

7.3.3 Primerjava vzporednih implementacij z razporejanjem podatkov in vzporednih implementacij z bitnimi rezinami

Primerjavo prepustnosti vzporednih implementacij z razporejanjem podatkov in vzporednih implementacij z bitnimi rezinami na platformi CUDA prikazujeta grafa na slikah 7.19 in 7.20. Iz grafov je razvidno, da imajo

implementacije z bitnimi rezinami manjšo prepustnost kot implementacije z razporejanjem podatkov.

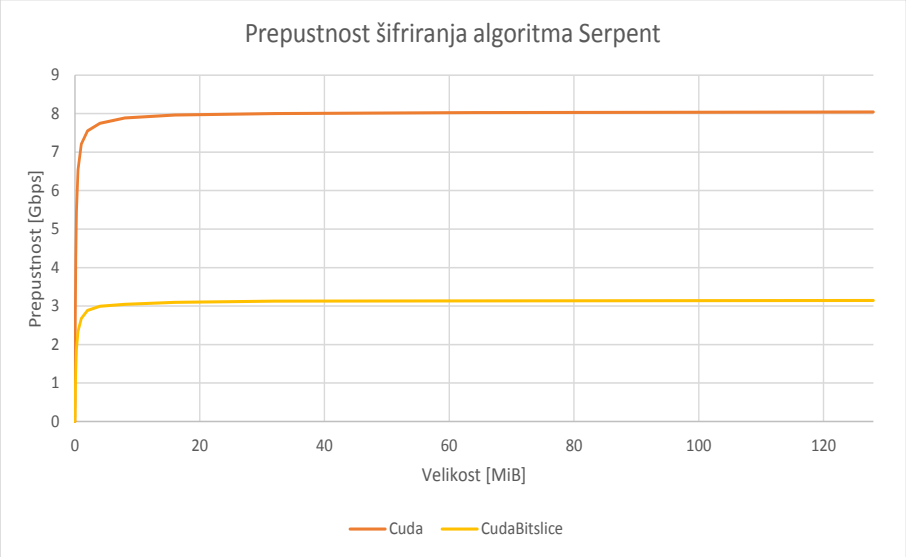


Slika 7.19: Primerjava prepustnosti implementacije z razporejanjem podatkov in z bitnimi rezinami algoritma Rijndael.

Implementacije z bitnimi rezinami imajo še vedno višjo prepustnost kot zaporedne implementacije, a dosežemo veliko manjšo pohitritev v primerjavi z implementacijami z razporejanjem podatkov. Pohitritev prikazuje graf na sliki 7.21.

Glavni razlog za tako slabo prepustnost je to, da je celotno šifriranje sestavljeno kar iz petih ščepcev: ustvarjanje števcev, transponiranje števcev, šifriranje, inverzno transponiranje in operacija XOR med šifriranimi števci in čistopisom. Delež časa, ki ga vzame posamezna operacija za oba algoritma prikazuje graf na sliki 7.22. Iz grafa je razvidno, da ostale operacije vzamejo približno polovico celotnega časa.

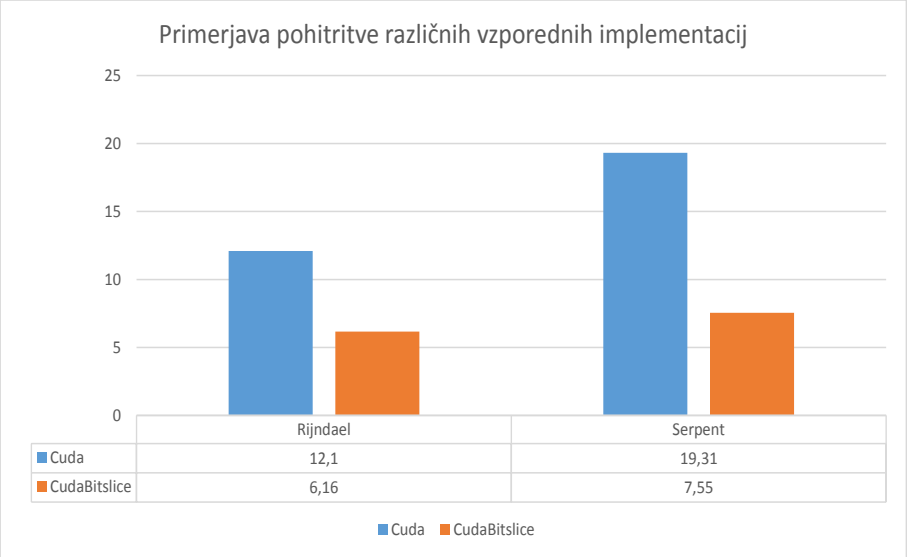
Če pa primerjamo samo šifrirne ščepce 7.23, ugotovimo da je v primeru algoritma Rijndael, šifrirni ščepec implementacije z bitnimi rezinami hitrejši kot pri implementaciji z razporejanjem podatkov. Do tega pride verjetno zato, ker pri bitnih rezinah obdelujemo 32 blokov naenkrat v načinu SIMD



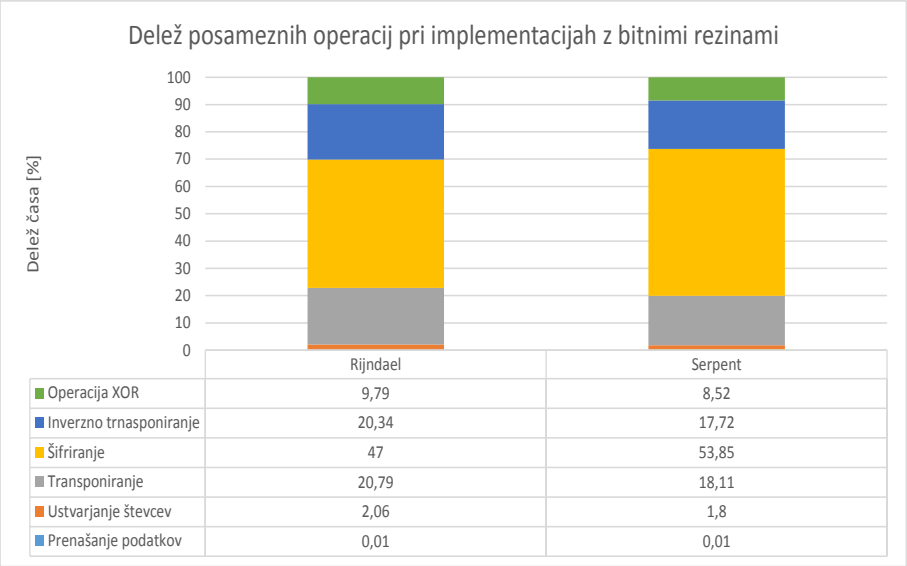
Slika 7.20: Primerjava prepustnosti implementacije z razporejanjem podatkov in z bitnimi rezinami algoritma Serpent.

s samo 16 niti. Zaradi tega vsaka nit povprečno naenkrat obdela dva bloka podatkov.

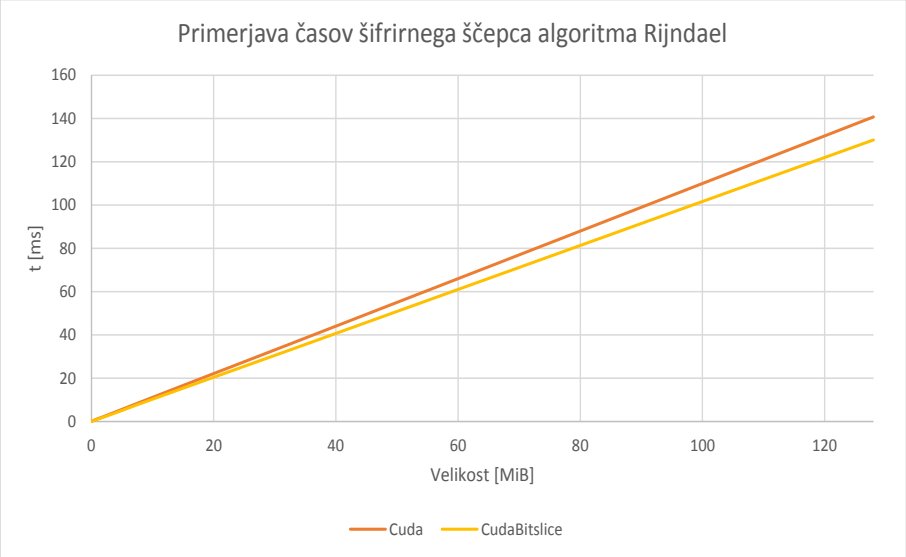
Po drugi strani je šifrirni ščepec algoritma Serpent z razporejanjem podatkov hitrejši kot z bitnimi rezinami 7.24. To je verjetno zaradi tega, ker smo bili pri implementaciji z bitnimi rezinami prisiljeni uporabiti deljeni pomnilnik za komunikacijo niti. Poleg tega pa smo morali posledično uporabiti sinhronizacijske točke za sinhronizacijo niti. Razlika med Serpentom in Rijndaelom je tudi to, da je razporeditev niti na blok enaka kot pri implementaciji z razporejanjem podatkov



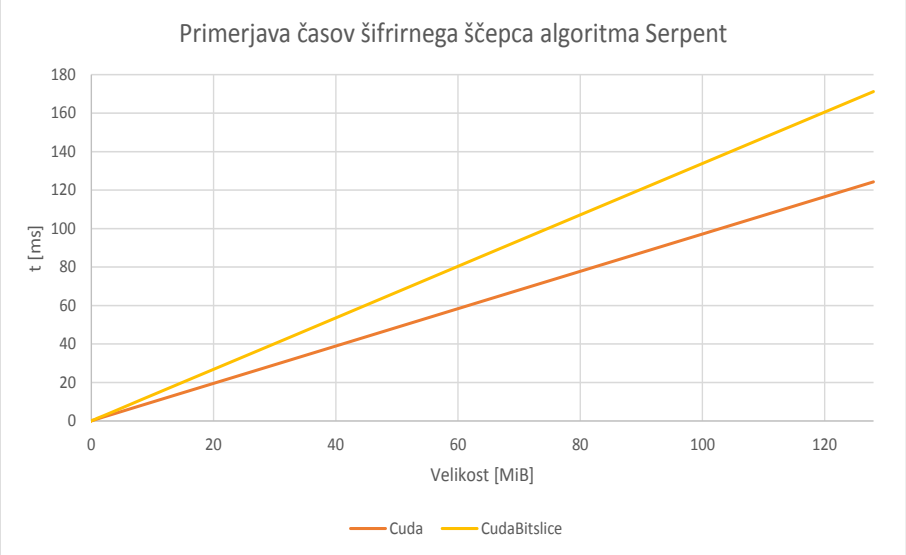
Slika 7.21: Primerjava pohitritve vzporedne implementacije z razporejanjem podatkov in z bitnimi rezinami.



Slika 7.22: Delež posameznih operacij implementacij z bitnimi rezinami.



Slika 7.23: Primerjava šifrirnega ščepca algoritma Rijndael.



Slika 7.24: Primerjava šifrirnega ščepca algoritma Serpent.

7.4 Sklep

Pri primerjavi zaporednih in vzporednih implementacij na osnovi razporejanja podatkov smo ugotovili, da nekatere vzporedne implementacije dosegajo tudi do dvajsetkratno pohitritev v primerjavi z zaporednimi. Najvišjo pohitritev sta dosegla algoritma RC6 in Serpent, ki ne uporabljata substitucijskih tabel shranjenih v deljenem pomnilniku, najslabšo pa Twofish zaradi velikega števila konfliktov v deljenem pomnilniku.

Poleg tega smo pri tej primerjavi ugotovili, da prepustnost vzporednih implementacij na začetku raste, nato pa se ustavi pri konstantni vrednosti. To se zgodi takrat, ko pride do zapolnitve sistema in ne moremo več obdelati vseh blokov naenkrat v konstantnem času.

Pri primerjavi implementacij narejenih na platformi CUDA in OpenCL smo ugotovili, da platforma OpenCL, kljub univerzalnosti še vedno dosega zadovoljive rezultate. Vseeno pa so bile hitrosti nekaterih algoritmov v primerjavi s platformo CUDA počasnejše tudi do 30 %.

Primerjava implementacij z razporejanjem podatkov in implementacij z bitnimi rezinami je pokazala, da so bile slednje počasnejše. Poglavitni razlog za to je bil, da so implementacije z bitnimi rezinami sestavljene iz dodatnih ščepcev, ki vzamejo približno 50 % celotnega časa. Primerjava samih ščepcev za šifriranje pa je pokazala, da je ta hitrejši pri implementaciji z bitnimi rezinami algoritma Rijndael. Ščepec za šifriranje algoritma Serpent v načinu bitnih rezin pa se je izkazal za počasnejšega, saj smo bili primorani uporabiti deljeni pomnilnik za komunikacijo med niti in sinhronizacijo niti.

Poglavje 8

Sklepne ugotovitve in nadaljnje delo

V magistrskem delu je bila narejena primerjava med optimalnimi zaporednimi in vzporednimi implementacijami petih finalistov AES. Zaporedne implementacije smo implementirali za CPE v programskem jeziku C in jih primerjali z vzporednimi implementacijami na osnovi razporejanja podatkov, ki smo jih implementirali za GPE na platformi CUDA. Poleg tega smo naredili tudi primerjavo med platformama CUDA in OpenCL. Prva je namenjena izključno za Nvidijine grafične kartice, druga pa za poganjanje programov na različnih vrstah GPE in večjedrnih procesorjev.

Poleg vzporednih implementacij z razporejanjem podatkov, kjer vsaka nit obdela svoj blok podatkov, smo primerjali tudi vzporedne implementacije z bitnimi rezinami na platformi CUDA. V tem načinu podatke najprej preoblikujemo tako, da so predstavljeni z bitnimi rezinami, kjer lahko z eno operacijo spremenimo bite več blokov naenkrat. Zaradi omejitev smo v tem načinu implementirali algoritma Rijndael in Serpent, ki pa sta bili prvi implementaciji bločnih šifer z bitnimi rezinami na platformi CUDA.

S testi smo pokazali, da so vzporedne implementacije z razporejanjem podatkov dosegle tudi do dvajsetkratno pohitritev v primerjavi z zaporednimi implementacijami. Pri tem smo ugotovili, da sta najvišjo pohitritev

dosegla algoritma Serpent in RC6, ki ne uporabljata deljenega pomnilnika za shranjevanje substitucijskih tabel. Algoritem Twofish je dosegel najmanjšo pohitritev, saj je prišlo do veliko konfliktov v deljenem pomnilniku.

Primerjava CUDE in OpenCL je pokazala, da je OpenCL počasnejši pri vseh petih algoritmih. Pri algoritmu RC6 celo do 30 %. Ne glede na svojo univerzalnost, je platforma OpenCL dosegla solidno prepustnost.

Primerjava implementacij z razporejanjem podatkov in z bitnimi rezinami je pokazala, da so slednje počasnejše. Pri algoritmu Rijndael se je izkazalo, da je težava to, da je celotno šifriranje sestavljeno iz številnih drugih ščepcev. Pri algoritmu Serpent pa je poleg tega bil problem tudi uporaba deljenega pomnilnika za komunikacijo niti in sinhronizacija. Vseeno smo naredili zanimivo raziskavo o tem, kako lahko bločne šifre implementiramo z bitnimi rezinami za arhitekturo CUDA.

Iz naših ugotovitev smo napisali tudi članek z naslovom "Parallel Bitslice AES-CTR implementation on CUDA", ki smo ga poslali v objavo v revijo Journal of Parallel and Distributed Computing [29].

V nadaljevanju bi lahko za začetek preizkusili implementacije pognati na novejših grafičnih karticah. Grafična kartica, s katero smo mi testirali implementacije, je v času preizkušanja že bila zastarela in niti ne najboljša. Pričakujemo lahko, da bi dosegli še višjo pohitritev.

Poleg tega bi lahko implementacije za platformo OpenCL preizkusili tudi na ostalih vzporednih arhitekturah kot so več jedrni procesorji in procesorji za obdelavo digitalnih signalov.

Navsezadnje bi lahko naš način razporejanja podatkov uporabili tudi s kakšno drugo šifro, ki deluje podobno kot Serpent in RC6. Ugotovili smo namreč, da je mogoče šifre, ki uporabljajo manj poizvedb v pomnilnik in več aritmetično-logičnih operacij, bolj pohitriti kot ostale.

Literatura

[1] D. L. Cook, J. Ioannidis, A. D. Keromytis in J. Luck, "Crypto-Graphics: Secret key cryptography using graphics cards", *Topics in Cryptology–CT-RSA 2005*, Springer, 2005, str. 334–350.

[2] O. Harrison in J. Waldron, *AES encryption implementation and analysis on commodity graphics processing units*, Springer , 2007.

[3] S. A. Manavski, "CUDA compatible GPU as an efficient hardware accelerator for AES cryptography", *2007 IEEE International Conference on Signal Processing and Communications (ICSPC 2007)*, 2007, str. 65–68.

[4] O. Harrison in J. Waldron, "Practical Symmetric Key Cryptography on Modern Graphics Hardware," *USENIX Security Symposium*, 2008, str. 195–209.

[5] A. Di Biagio, A. Barenghi, G. Agosta in G. Pelosi, "Design of a parallel AES for graphics hardware using the CUDA framework", *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS 2009)* , 2009, str. 1–8.

[6] K. Iwai, T. Kurokawa in N. Nisikawa. "AES encryption implementation on CUDA GPU and its analysis", *2010 First International Conference on Networking and Computing (ICNC)*, 2010, str. 209–214.

[7] C. Mei, H. Jiang in J. Jenness. "CUDA-based AES parallelization with fine-tuned GPU memory utilization", *P2010 IEEE International Sym-*

posium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010, str. 1–7.

[8] Q. Li, C. Zhong, K. Zhao, X. Mei in X. Chu, "Implementation and Analysis of AES Encryption on GPU", *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012, str. 843–848.

[9] N. Nishikawa, K. Iwai in T. Kurokawa, "High-performance symmetric block ciphers on cuda", *2011 Second International Conference on Networking and Computing (ICNC)*, 2011, str. 221–227.

[10] X. Wang, X. Li, M. Zou in J. Zhou, "AES finalists implementation for GPU and multi-core CPU based on OpenCL", *2011 IEEE International Conference on Anti-Counterfeiting, Security and Identification (ASID)*, 2011, str. 38–42.

[11] E. Al Shamsi in A. Mohamed, "Block Ciphers on GPU: Integration and Evaluation of the improvement", ENSIMAG, France, Tehnično poročilo, 2009.

[12] A. M. Nazlee, F. A. Hussin in N. B. Z. Ali, "Serpent encryption algorithm implementation on compute unified device architecture (cuda)", *2009 IEEE Student Conference on Research and Development (SCOReD)*, 2009 str. 164–167.

[13] E. Biham, "A Fast New DES Implementation in Software", *Fast Software Encryption*, Springer, 1997, str. 260–272.

[14] C. Rebeiro, D. Selvakumar in A. S. L. Devi, "Bitslice implementation of AES" *Cryptology and Network Security*, Springer, 2006, str. 203–212.

[15] D. R. Stinson, "Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications)". Chapman and Hall/CRC, 2005.

[16] R. Anderson, E. Biham in L. Knudsen, "Serpent: A proposal for the advanced encryption standard". *NIST AES Proposal*, zv. 174, 1998.

[17] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall in N. Ferguson, "Twofish: A 128-bit block cipher." *NIST AES Proposal*, zv. 15, 1998.

[18] R. L. Rivest, "The RC5 encryption algorithm", *Fast Software Encryption*, Springer, 1995, str. 86–96.

[19] R. L. Rivest, M. Robshaw, R. Sidney in Y. L. Yin, "The RC6TM block cipher", *First Advanced Encryption Standard (AES) Conference*, 1998.

[20] J. Nechvatal, E. Barker, L. Bassham, W. Burr in M. Dworkin, "Report on the development of the Advanced Encryption Standard (AES)", DTIC Document, Tehnično poročilo, 2000.

[21] C. Burwick, D. Coppersmith, E.q D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. M. Matyas Jr, L. O'Connor, M. Peyravian, D. Safford in dr., "MARS-a candidate cipher for AES", *NIST AES Proposal*, zv. 268, 1998.

[22] S. Cook, "CUDA programming: a developer's guide to parallel computing with GPUs", Newnes, 2013.

[23] R. Banger in K. Bhattacharyya, "OpenCL Programming by Example", Packt Publishing Ltd, 2013.

[24] D. A. Osvik, "Speeding up Serpent", *AES Candidate Conference*, 2000, str. 317–329.

[25] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques", DTIC Document, Tehnično poročilo, 2001.

[26] P. Maistri, F. Masson in R. Leveugle, "Implementation of the Advanced Encryption Standard on GPUs with the NVIDIA CUDA framework" *2011 IEEE Symposium on Industrial Electronics and Applications (ISIEA)*, 2011, str. 213–217.

[27] V. Rijmen, "Efficient Implementation of the Rijndael S-box", *Katholieke Universiteit Leuven, Dept. ESAT. Belgium*, 2000.

[28] D. Canright, "A very compact S-box for AES", *Cryptographic Hardware and Embedded Systems (CHES 2005)*,2005, str. 441–455.

[29] K. Zupan, T. Dobravec, "Parallel Bitslice AES-CTR implementation on CUDA", *Poslano v objavo v Journal of Parallel and Distributed Computing*, Junji 2015.

[30] RSA Laboratories, "Has DES Been Broken?". Donstopno na: `http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/has-des-been-broken.htm`. Zadnjič dostopano: 2015.

[31] Specification for the advanced encryption standard (aes). Dostopno na: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`. Zadnjič dostopano: 2015.

[32] Porting CUDA Applications to OpenCL. Dostopno na: `http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/programming-in-opencl/porting-cuda-applications-to-opencl/`. Zadnjič dostopano: 2015.

[33] Wikipedia, "Advanced Encryption Standard". Dostopno na: `https://en.wikipedia.org/wiki/Advanced_Encryption_Standard`. Zadnjič dostopano: 2015.

[34] Wikipedia, "Serpent (cipher)". Dostopno na: `https://en.wikipedia.org/wiki/Serpent_%28cipher%29`. Zadnjič dostopano: 2015.

[35] Wikipedia, "Feistel cipher". Dostopno na: `https://en.wikipedia.org/wiki/Feistel_cipher`. Zadnjič dostopano: 2015.

[36] Wikipedia, "Twofish". Dostopno na: `https://en.wikipedia.org/wiki/Twofish`. Zadnjič dostopano: 2015.

[37] Serpent S Boxes as Boolean Functions. Dostopno na: `http://brgladman.org/oldsite/cryptography_technology/serpent/index.php`. Zadnjič dostopano: 2015.